TAL TECH

DOCTORAL THESIS

Machine Learning-Based Detection and Characterization of Evolving Threats in Mobile and IoT Systems

Alejandro Guerra Manzanares

TALLINNA TEHNIKAÜLIKOOL
TALLINN UNIVERSITY OF TECHNOLOGY
TALLINN 2022

TALLINN UNIVERSITY OF TECHNOLOGY
DOCTORAL THESIS
54/2022

Machine Learning-Based Detection and Characterization of Evolving Threats in Mobile and IoT Systems

ALEJANDRO GUERRA MANZANARES

TAL TECH PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

The dissertation was accepted for the defence of the degree of Doctor of Philosophy in Computer Science (Cybersecurity) on 12 August 2022

Supervisor: Research Professor Dr. Hayretdin Bahsi,
Department of Software Science, School of Information Technologies,
Tallinn University of Technology,
Tallinn, Estonia

Co-supervisor: Senior Researcher Dr. Sven Nõmm,
Department of Software Science, School of Information Technologies,
Tallinn University of Technology,
Tallinn, Estonia

Co-supervisor: Assistant Professor Dr. Marcin Luckner,
Faculty of Mathematics and Information Science,
Warsaw University of Technology,
Warsaw, Poland

Opponents: Professor Dr. Juan Manuel Corchado Rodriguez,
Department of Computer Science and Automation Control,
University of Salamanca,
Salamanca, Spain

Professor Dr. Ali Akbar Ghorbani,
Faculty of Computer Science,
University of New Brunswick,
Fredericton, Canada

Defence of the thesis: 1 September 2022, Tallinn

Declaration:
Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology, has not been submitted for any academic degree elsewhere.

Alejandro Guerra Manzanares _____
signature

ISSN 2585-6898 (publication)
ISBN 978-9949-83-898-1 (publication)
ISSN 2585-6901 (PDF)
ISBN 978-9949-83-899-8 (PDF)
Printed by Koopia Niini & Rauam

TALLINNA TEHNIKAÜLIKOOL
DOKTORITÖÖ
54/2022

Masinõppepõhine arenevate ohtude tuvastamine ning kirjeldamine mobiilseadmete ja värkvõrkude jaoks

ALEJANDRO GUERRA MANZANARES

TAL TECH
KIRJASTUS

Contents

List of publications

The present Ph.D. thesis is based on the following publications that are referred to in the text by Roman numbers.

I A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. In-depth feature selection and ranking for automated detection of mobile malware. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*, pages 274–283. INSTICC, SciTePress, 2019

II A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Differences in android behavior between real device and emulator: A malware detection perspective. In *Proceedings of the 6th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 399–404, 2019

III A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. Time-frame analysis of system calls behavior in machine learning-based mobile malware detection. In *International Conference on Cyber Security for Emerging Technologies (CSET)*, pages 1–8, 2019

IV A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Kronodroid: Time-based hybrid-featured dataset for effective android malware detection and characterization. *Computers & Security*, 110:102399, 2021

V A. Guerra-Manzanares, M. Luckner, and H. Bahsi. Android malware concept drift using system calls: Detection, characterization and challenges. *Expert Systems with Applications*, 206:117200, 2022

VI A. Guerra-Manzanares, M. Luckner, and H. Bahsi. Concept drift and cross-device behavior: Challenges and implications for effective android malware detection. *Computers & Security*, 120:102757, 2022

VII A. Guerra-Manzanares and M. Välbe. Cross-device behavioral consistency: Benchmarking and implications for effective android malware detection. *Machine Learning with Applications*, 9:100357, 2022

VIII A. Guerra-Manzanares, H. Bahsi, and M. Luckner. Leveraging the first line of defense: A study on the evolution and usage of android security permissions for enhanced android malware detection. *Journal of Computer Virology and Hacking Techniques*, *in press*, 2022

IX A. Guerra-Manzanares and H. Bahsi. On the relativity of time: Implications and challenges of data drift on long-term effective android malware detection. *Computers & Security*, *in press*:102835, 2022

X A. Guerra-Manzanares and H. Bahsi. On the application of active learning to handle data evolution in android malware detection. *Conference paper*, *under review*, 2022

XI A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Hybrid feature selection models for machine learning based botnet detection in iot networks. In *2019 International Conference on Cyberworlds (CW)*, pages 324–327, 2019

XII A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. Towards the integration of a post-hoc interpretation step into the machine learning workflow for iot botnet detection. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 1162–1169, 2019

XIII A. Guerra-Manzanares, J. Medina-Galindo, H. Bahsi, and S. Nõmm. Medbiot: Generation of an iot botnet dataset in a medium-sized iot network. In *Proceedings of the 6th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*, pages 207–218, 2020

XIV A. Guerra-Manzanares, J. Medina-Galindo, H. Bahsi, and S. Nõmm. Using medbiot dataset to build effective machine learning-based iot botnet detection systems. In *International Conference on Information Systems Security and Privacy*, pages 222–243. Springer, 2020

XV A. Guerra-Manzanares and H. Bahsi. On the application of active learning for efficient and effective early iot botnet detection. *Journal article, under review*, 2022

Author's contributions to the publications

I In Publication I, I was the main author, designed and carried out all the experimentation, collected and processed the data, built the ML models, analyzed the results, prepared the figures, and wrote the manuscript.

II In Publication II, I was the main author, designed and carried out all the experimentation, collected and processed the data, built the ML models, analyzed the results, prepared the figures, and wrote the manuscript.

III In Publication III, I was the main author, designed and carried out all the experimentation, collected and processed the data, built the ML models, analyzed the results, prepared the figures, and wrote the manuscript.

IV In Publication IV, I was the main author, designed and carried out all the experimental workflow, collected and processed all the acquired data, generated the data sets, analyzed the data sets, prepared the figures, and wrote the manuscript.

V In Publication V, I was the main author, wrote the simulation program, carried out the simulations, proposed modifications to the original algorithm, analyzed the results, prepared the figures, and wrote the manuscript.

VI In Publication VI, I was the main author, wrote the simulation program, carried out the simulations, analyzed the results, prepared the figures, and wrote the manuscript.

VII In Publication VII, I was the main author, I supervised the workflow for the data acquisition phase and posterior statistical analysis, prepared the data for ML tasks, built all the ML models, analyzed the results, prepared the figures, and wrote the manuscript.

VIII In Publication VIII, I was the main author, wrote the simulation program, carried out the simulations, analyzed the results, prepared the figures, and wrote the manuscript.

IX In Publication IX, I was the main author, wrote the simulation program, carried out the simulations, analyzed the results, prepared the figures, and wrote the manuscript.

X In Publication X, I was the main author, wrote the simulation program, carried out the simulations and the analysis of the results, prepared the figures, and wrote the manuscript.

XI In Publication XI, I was the main author, wrote the simulation program, carried out the simulations and the analysis of the results, prepared the figures, and wrote the manuscript.

XII In Publication XII, I was the main author, prepared the data for ML tasks, built all the ML models, analyzed the results, prepared the figures, and wrote the manuscript.

XIII In Publication XIII, I was the main author, built all the ML models, analyzed the results, prepared the figures, and wrote the manuscript.

XIV In Publication XIV, I was the main author, wrote the simulation program, carried out the simulations and the analysis of the results, prepared the figures, and wrote the manuscript.

XV In Publication XV, I was the main author, wrote the simulation program, carried out the simulations and the analysis of the results, prepared the figures, and wrote the manuscript.

Abbreviations

AI	Artificial Intelligence
App	Application
AV	Antivirus
C&C	Command & Control
DDoS	Distributed Denial-of-Service
DES	Dynamic Ensemble Selection
DSEL	Dynamic Selection Dataset
DT	Decision Tree
FS	First Seen timestamp
IDS	Intrusion Detection System
IoT	Internet of Things
k-NN	k-Nearest Neighbors
LM	Last Modification timestamp
Malware	Malicious software
ML	Machine Learning
OEM	Original Equipment Manufacturer
OS	Operating System
RF	Random Forest
SOC	Security Operations Center
SVM	Support Vector Machines
Tbps	Terabit per second
XAI	eXplainable Artificial Intelligence

"Life is what happens when you're busy making other plans"
- John Lennon

This doctoral dissertation, written by a former marine who dreamed about being a pilot and never planned to pursue a college degree, is a vivid example of how surprising and challenging life can be.

Amor fati. Elu on ilus

"Who dares wins"
- motto of the Special Air Service (SAS)

1 Introduction

We live in an interconnected world. The internet has been a disruptive technology that has enabled the interconnection and intercommunication of people and computers worldwide. However, the tremendous advances on data-based communication have generated new threats and security challenges for individuals and organizations. Malicious software, shortened as *malware*, is one of the most prominent threats. This dissertation explores the application of machine learning techniques to tackle relevant cyber security issues related to two highly targeted systems by malware, namely, mobile devices and internet-of-things devices. In the mobile domain, malware has been used to infiltrate mobile phones with the intention of stealing, hijacking, or corrupting the wealth of *sensitive* data that these devices store to provide immediate financial gain for the attackers (e.g., sending premium SMS messages). In the Internet of Things (IoT) domain, malware has been used to compromise IoT devices with the primary objective of using them to perpetrate large-scale cyber attacks (e.g., the record-breaking distributed denial-of-service (DDoS) attack of 3.47 Tbps targeting an *Azure* customer in Asia [95]). In both cases, the attackers exploit the systems' capabilities for their benefit, generating direct revenue for them or substantial losses for the targeted entity.

Due to the constant evolution of the threat landscape, the traditional countermeasures against malware, such as signature-based detection systems, have become ineffective, unable to catch up with malware developments, especially with *zero-day*, *obfuscated*, and *evolved* malware. To address this issue, machine learning techniques have been explored in both cyber domains with remarkable success.

Mobile malware focuses mainly on Android devices due to the open nature of the system and the large target audience who use Android devices. In this regard, ML-based malware detection solutions for mobile devices have been proposed since the initial development of the Android OS. Using a wide variety of techniques, the vast majority of these detection models are induced with *static* snapshots of historical data aiming to detect future malware. These *static* models assume that the underlying properties of the data distribution do not change over time. However, the threat landscape targeting Android devices is dynamic, and ever-evolving since the early days of the popular mobile operating system. Malware families evolve in a constant spiral of sophistication, revolving around the large attack surface exposed by mobile devices. This shift in the threat landscape presents detection models with an expiration date, and they may become obsolete over time should adaptive actions not be taken to address the underlying data changes, a phenomenon called *concept drift*. Despite this fact, most proposed solutions neglect concept drift and its degenerative impact on the learning models over time. Addressing this significant research gap, concept drift handling and its characterization for effective long-time detection performance of Android malware are the main issues tackled in this dissertation. Besides, the small proportion of studies that addressed concept drift issues in Android malware detection, did not investigate the impact of different timestamps on the data modeling and the overall performance of the learning models over time. Timestamp selection, a critical component for effective data modeling and concept drift handling, that has not previously been addressed in the related research, is comprehensively examined in this dissertation. Lastly, Android malware research using behavioral data (i.e., dynamic features) is based on the assumption that the behavior of the apps is consistent across devices, thus explaining the heterogeneity of different configurations of devices and operating system versions found in research studies and enabling the generalization of their systems to *any* Android device data. Simply put, it is assumed that the nature of the devices (i.e., real device or emulator) and the OS versions used do not matter and

that the logged data is consistent across device configurations, allowing effective cross-device malware detection. However, the results of [3] that considered different kinds of devices in the same setting challenge this assumption. In this regard, the validity of the cross-device behavioral consistency and its impact on the ML-based detection models is explored in this dissertation for system calls, the most commonly used dynamic feature set in Android detection systems [70].

On the other hand, IoT malware is characterized by using more simplistic yet effective attack vectors. Due to the lack of security measures and proper management that characterize IoT devices, their compromise is usually achieved through the exploitation of well-known but not patched vulnerabilities or the usage of default *admin* credentials. Once compromised, the *bot* is used to amplify cyber attacks over the network which may cause significant financial and reputation losses. ML-based intrusion detection systems have been proposed to detect and mitigate the generated attacks. However, despite their effectiveness, these *reactive* measures can still yield significant losses for the targeted system. Early identification of botnet formation prior to attack delivery (i.e., spreading and C&C communication) and awareness of the main attack aspects could be valuable in this respect to improve the detection models employed in such resource-constrained devices and prevent attacks from occurring. These research gaps and areas not thoroughly explored in the related IoT botnet literature are addressed in this dissertation.

The objectives of this doctoral dissertation and its main contributions to the field are provided in the following sections.

1.1 Research objectives

This thesis is composed of two parts with different but related research objectives. In both cases, for Android malware detection, as well as for IoT botnet detection, the underlying research objective is to enhance the effectiveness and efficiency of the machine learning-based detection systems designed for such tasks. However, specific research objectives are defined for each application domain. They are described in the subsequent paragraphs.

1.1.1 Mobile malware detection objectives

The specific research objectives related to mobile malware detection tackled by this thesis are defined as follows:

- **RO1**: Generation of a data set suitable for concept drift and cross-device detection issues exploration.
- **RO2**: Demonstration and characterization of concept drift in Android data *dynamic* (system calls) and *static* (permissions) feature spaces.
- **RO3**: Application of an ML-based solution to handle effectively concept drift for Android malware detection.
- **RO4**: Evaluation of different timestamps for effective concept drift handling and modeling.
- **RO5**: Assessment of the validity of the cross-device postulate and its implications for effective detection.

1.1.2 IoT botnet detection objectives

The specific research objectives related to IoT botnet detection guiding the work performed in this thesis are defined as follows:

- **RO6**: Evaluation of feature selection techniques and post-hoc interpretation methods for enhanced attack detection.
- **RO7**: Application of supervised and unsupervised ML-based methods for *early* IoT botnet detection.
- **RO8**: Evaluation of active learning strategies for early IoT botnet detection.

1.2 Contribution to the field

This thesis is based on a collection of peer-reviewed scientific publications published in reputed scientific journals and international conferences. The primary objective of this thesis is to enhance the performance in terms of effectiveness and efficiency of the machine learning-based detection systems designed for Android malware detection and IoT botnet detection tasks. In this regard, the main contributions of this dissertation to the cyber security field are:

1. The generation of *KronoDroid*, a novel, and publicly available data set that enables the exploration of concept drift and cross-device detection issues for Android malware detection for the whole Android historical timeline (i.e., since 2008).
2. The thorough statistical analysis of the differential features and discriminatory factors between benign and malware Android applications.
3. The demonstration and characterization of concept drift in Android malware detection for distinct feature spaces (i.e., system calls, permissions and *static* API calls).
4. The comparison and thorough evaluation of the impact of timestamps as key factors to model and handle concept drift effectively.
5. The demonstration that cross-device behavioral consistency can not be assumed for system calls-based Android applications data.
6. The enhancement of an automated method to handle concept drift effectively for Android malware detection in different feature spaces, yielding long-term high detection performance and robustness against concept drift and imbalanced data issues.
7. The application and demonstration of the benefits of the active learning approach to concept drift handling for efficient and effective Android malware detection.
8. The generation of guidelines and recommendations to design enhanced Android malware detection systems.
9. The application of feature selection techniques and interpretation methods for a better understanding of the phenomenon and induction of enhanced ML-based models for attack detection in the IoT botnet domain.
10. The demonstration that *MedBIoT* data set can be used to detect IoT botnet formation at early stages of botnet deployment, thus preventing the nefarious consequences of IoT-based attacks.

11. The application and demonstration of the benefits of the active learning approach for early IoT botnet detection in SOC environments.

For the sake of clarity, Table 1 provides the mapping of the chapters of this document to the corresponding research objectives, publications, and contributions of this dissertation.

Table 1: Mapping among thesis chapters, research objectives, publications, and contributions to the field of this doctoral dissertation

Chapter	*Research Objectives*	*Publications*	*Contributions*
5	RO1	I, II, III, IV	1, 2
6	RO2, RO3, RO4	V	3, 4, 6
7	RO2, RO3, RO5	VI	5, 8
8	RO5	VII	5
9	RO2, RO4	VIII	3, 4
10	RO4	IX	3, 4
11	RO3	X	7
12	RO6	XI, XII	9
13	RO7, RO8	XIII, XIV, XV	10, 11

1.3 Thesis structure

This thesis is divided into 14 chapters and split into two parts, i.e., Part I and Part II, covering 2 research topics. Briefly summarized, after four general introductory chapters, Part I, *about Android malware detection*, encompasses Chapters 5 to 11, whereas Part II, *about IoT botnet detection*, includes Chapters 12 and 13. Chapter 14 summarizes the main conclusions and provides future work.

A brief description of the content of each chapter is provided as follows.

- **Chapter 1**, *Introduction*, provides a brief overview of the problem statement, the research objectives and the main contributions of this research work.
- **Chapter 2**, *Learning under concept drift*, provides background information about the impact of concept drift in learning models, its formal definition and main typologies.
- **Chapter 3**, *Dissecting Android malware detection*, provides the main characteristics of Android malware detection research in the context of observed research gaps and related works.
- **Chapter 4**, *A brief on IoT botnet detection*, provides background information about IoT botnets and a summary of related works in the IoT botnet detection literature.

Part I, *About Android Malware Detection*.

- **Chapter 5**, *KronoDroid: a historical Android data set*, describes the content of the works that lead to the generation of *KronoDroid* and the main features, methodological workflow and analysis of the *KronoDroid* data set.
- **Chapter 6**, *Concept drift on behavioral data: detection, handling, and characterization*, demonstrates and characterizes concept drift for Android behavioral data (i.e., system calls) and proposes a method to handle it effectively.

- **Chapter 7**, *Concept drift and cross-device behavior: Implications for effective detection*, explores the combination of two of the main challenges for effective long-term Android malware detection, namely, concept drift and cross-device behavioral detection.

- **Chapter 8**, *Cross-device behavioral consistency: Benchmarking and implications for effective detection*, describes and reports the results of the benchmarking setup that enabled the testing of the cross-device behavioral consistency postulate on a varied set of Android devices.

- **Chapter 9**, *Leveraging the first line of defense against malware: Android security permissions*, tackles and characterizes concept drift and malware family evolution in the permissions feature space.

- **Chapter 10**, *On the relativity of time: A study of timestamps for effective Android concept drift handling*, compares and describes the impact of different timestamps on Android malware detection under concept drift constraints.

- **Chapter 11**, *Applying active learning to handle data evolution in Android malware detection*, explores the usage of the active learning approach to deal with concept drift for long-term effective Android malware detection.

Part II, *About IoT Botnet Detection.*

- **Chapter 12**, *IoT botnet attack detection*, evaluates the usage of feature selection techniques and interpretation methods to enhance IoT botnet attack detection systems.

- **Chapter 13**, *IoT botnet attack prevention*, explores the usage of *MedBioT* data set for early stage IoT botnet detection and the active learning approach as a means to prevent IoT botnet-based attacks.

- **Chapter 14**, *Conclusions and future work*, provides the main conclusions and outlines future work.

2 Learning under concept drift

This section provides background information about concept drift, its formal definition, and its main typologies.

2.1 Static models for a dynamic world

The vast majority of learning models are *static*. That is, they are constructed under the assumption that the input data is steady and consistent over time (i.e., *stationary* data); as a result, learning models are constructed using previous data with the goal of effectively coping with future data. These models are generated to reflect the problem domain as it was then the model was formed, and not expecting significant changes or variability over time [102]. However, the world is dynamic, and non-stationary data distributions can be found in many problem domains. The evolution of data over time adds additional complexity to the data modeling process that, if not addressed, might impact the generalization capabilities of the models induced and their performance on future data. The critical impact of the temporal dimension on the learning models can lead to model obsolescence over time. Due to the dynamic development of cyber attacks brought on by the continuous conflict between attackers and defenders, the corresponding detection models are susceptible to concept drift issues.

2.2 Concept drift definition

Concept drift is the phenomenon in which the statistical properties of relevant data in a problem domain change over time in an unforeseeable way [71]. Formally, concept drift can be defined as follows:

Given a time period $[t_0, t_1]$ and a set of data instances belonging to that period $S_{t_0,t_1} = \{d_{t_0}, ..., d_{t_1}\}$, where $d_i = (x_i, y_i)$ is a single observation, $x_i = (x_i^1, x_i^2, \ldots, x_i^n) \in \mathbf{X}$ is the feature vector, $y_i \in \mathbf{Y}$ is the target label, and S_{t_0,t_1} follows a certain distribution $F_{t_0,t_1}(\mathbf{X}, \mathbf{Y})$. In this setting, concept drift is observed at t_2 if $F_{t_0,t_1}(\mathbf{X}, \mathbf{Y}) \neq F_{t_2,\infty}(\mathbf{X}, \mathbf{Y})$, and denoted as $\exists t : P_t(\mathbf{X}, \mathbf{Y}) \neq P_{t+1}(\mathbf{X}, \mathbf{Y})$ [71].

Based on this definition, concept drift at time period t_i can be equated as the change in the joint probability of $\mathbf{X}$ and $\mathbf{Y}$ at time t_i, expressed as $P_{t_i}(\mathbf{X}, \mathbf{Y})$. As the joint probability can be decomposed in two components, $P_{t_i}(\mathbf{X}, \mathbf{Y}) = P_{t_i}(\mathbf{X}) \times P_{t_i}(\mathbf{Y} \mid \mathbf{X})$, concept drift can be originated from three sources [71]:

1. $P_t(\mathbf{X}) \neq P_{t+1}(\mathbf{X})$ and $P_t(\mathbf{Y} \mid \mathbf{X}) = P_{t+1}(\mathbf{Y} \mid \mathbf{X})$. In this case, there is a shift in the data distribution, $P_t(\mathbf{X})$, that has no impact on the posterior probability, $P_t(\mathbf{Y} \mid \mathbf{X})$, thus not affecting the decision boundary of the model. This phenomenon is named *virtual concept drift* and it is depicted as *case 1* in Figure 1.

2. $P_t(\mathbf{X}) = P_{t+1}(\mathbf{X})$ and $P_t(\mathbf{Y} \mid \mathbf{X}) \neq P_{t+1}(\mathbf{Y} \mid \mathbf{X})$. In this case, the data distribution remains unchanged but the drift in the posterior probability will change the decision boundary and lead to a decrease in learning accuracy. This phenomenon is named *real concept drift* and it is depicted as *case 2* in Figure 1.

3. $P_t(\mathbf{X}) \neq P_{t+1}(\mathbf{X})$ and $P_t(\mathbf{Y} \mid \mathbf{X}) \neq P_{t+1}(\mathbf{Y} \mid \mathbf{X})$. In this case, the change in the data distribution (i.e., virtual concept drift) coexists with a change in the decision boundary (i.e., real concept drift). As these changes affects the decision boundary, this phenomenon is also reflected as *real concept drift* and it is depicted as *case 3* in Figure 1.

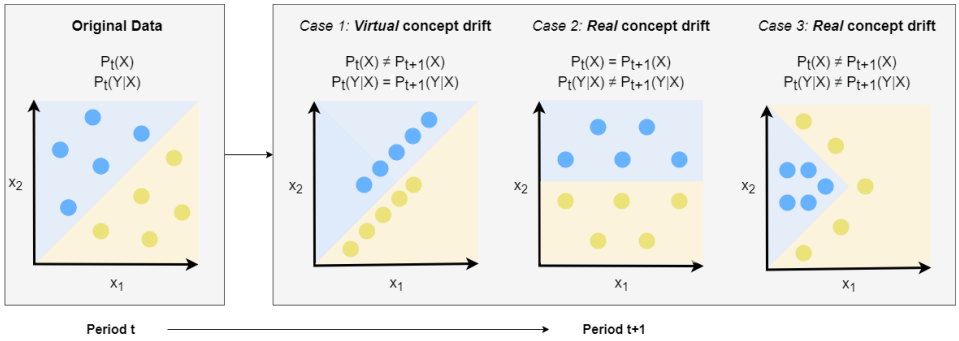


Figure 1: Sources of concept drift

As can be observed in Figure 1, only the real concept drift changes the decision boundary of the learning model provoking the obsolescence of the previous model. *Real concept drift* refer to changes in $P(\mathbf{Y} \mid \mathbf{X})$, which may happen with or without a change in $P(\mathbf{X})$ [32]. In *virtual concept drift*, also called *feature space drift* or *covariate shift*, the underlying data distribution is changed without affecting $P(\mathbf{Y} \mid \mathbf{X})$. From a predictive perspective, only the shift that affects the decision boundary, i.e., prediction decision, requires the adoption of adaptive measures [32].

2.3 Types of concept drift

The changes in data distribution leading to concept drift can occur in different forms and speeds over time. Figure 2 illustrates in a one-dimensional toy example the four main typologies of concept drift [71] and the *outliers* case (i.e., blips), which is usually not considered *true* concept drift but can present a remarkable challenge for proper concept drift handling [32, 83].

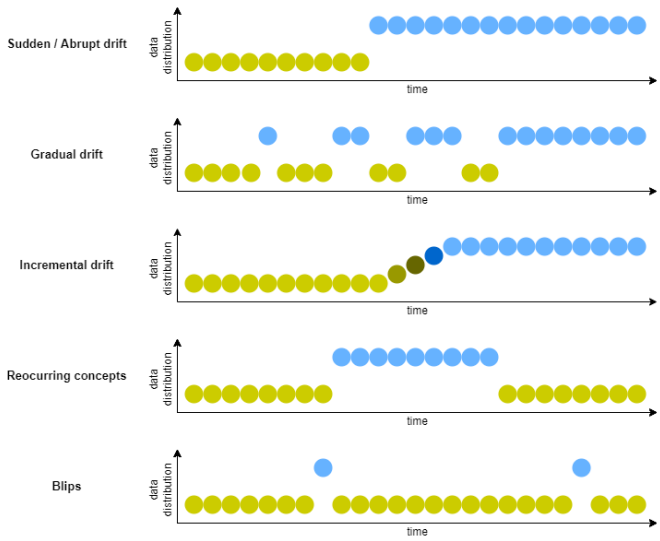


Figure 2: Types of concept drift patterns and the outliers case, based on [32, 71, 83]

The basic concept drift typologies, depicted in Figure 2, are briefly described in the

following points.

- *Sudden drift*: the switch to a new concept occurs abruptly, within a short period of time.
- *Incremental drift*: an old concept changes slowly over time through many intermediate concepts before reaching a new concept.
- *Gradual drift*: a new concept gradually replaces an old one over a period of time. In the transition phase both concepts coexist.
- *Reocurring concepts*: historical, previously seen concepts may reoccur after some time.
- *Blips*: singular random deviations or *outlier* cases appear in the data distribution.

A combination of concept drift typologies is frequently observed in practice, even while adaptive learning solutions designed to address concept drift issues may explicitly or implicitly specialize in particular subsets of concept drift, such as sudden non-reoccurring drifts [32]. Additionally, concept drift might last for an extended period and may not occur at a particular moment. In this case, intermediate concepts may arise during the transition between the old concept and the new concept. The intermediate concept can be interpreted as a mixture of both concepts, like in the incremental drift, or one of the old or the new concept, as in the gradual drift [71]. In general, *blips* are not considered concept drift, thus no adaptive measures should be taken to address such cases [32].

2.4 The impact of concept drift

Predictive models under concept drift influence require the implementation of adaptive solutions to detect and react to the changing environment, otherwise, their performance will degrade over time. As time goes on, the knowledge of the decision model might need to be updated including the new data, or be completely replaced to address the new scenario [32]. If no adaptive measure is taken, the model might become obsolete, and completely ineffective in dealing with the new incoming data.

In general, concept drift methods should be able to detect concept drift as it occurs and perform an adaptive change according to the severity and region of the drift to handle the new concept properly [71]. Detecting and understanding concept drift, including knowing where, how, and where the drift occurred is essential information that is used as input to implement *drift adaptation* solutions that extend the life of the machine learning-based solution. In this regard, the generation of a new model to substitute an obsolete model when *global* drift is detected (i.e., concept drift affecting most of the feature space), the usage of a model ensemble for recurring concept drifts, and the adjustment of existing models to address *regional* drift (i.e., concept drift is localized only in a limited area of the whole feature space) are common approaches to handle emerging concept drift [71].

3 Dissecting Android malware detection

Machine learning-based Android malware detection models are built using *static* and *dynamic* features extracted from Android applications (i.e., *apk* files) [70].

Static features are acquired without executing the app, generally extracted from the source code or the *apk* bundle. Features such as security permissions, API calls, intents, and activities, which can be collected using static analysis tools, fall within this category. In general, static features are swift and easy to collect in an automated fashion. However, the detection systems built based on these features are prone to be bypassed by *zero-day* and sophisticated malware, especially when obfuscation and encryption techniques are used.

The extraction of the so-called *dynamic* features requires the app to be installed and executed, enabling the capture of the *real* behavior of the running app in a *live* environment. Features such as system calls and network flow data are collected using this approach. In general, the acquisition of dynamic features is a time-consuming and challenging task. However, they tend to generate more robust and effective detection systems.

There exists a large body of research related to Android malware detection [70]. A myriad of methods, ranging from formal verification methods to deep learning, have been proposed to generate robust and effective detection systems. The related literature in the field reports high performance metrics to support the suitability of their approaches to tackle the Android malware detection task effectively. Therefore, from a broad perspective, the Android malware detection task may seem an already *solved* issue from the machine learning challenge point of view. However, a closer inspection of the state-of-the-art techniques evidences remarkable research gaps. These issues are briefly explained in the following sections and constitute the core research performed leading to this dissertation.

3.1 Neglecting the impact of time: concept drift

The vast majority of ML-based Android malware detection solutions proposed in the related literature are *static* models built using *snapshots* of data from the Android historical timeline, usually using the same data sets. *MalGenome* [113] and *Drebin* [11] are the most used data sets for Android malware research. Despite their small size and containing *outdated* data (i.e., both were collected before 2013), they have been utilized as the main sources of malware samples in the related literature, and even in recent publications. Even though some studies complemented these data with more recent and larger data sets, such as *Android Malware Dataset* (AMD) [103], to mitigate data-related issues [54] and increase the representativeness of the data set, they still relied on incomplete, relatively *old*, and *short* snapshots of malware data from the entire Android historical timeline, which ranges from 2008 to the present date. In addition to the *static* data issue, the common practice used in machine learning to handle data sets that consists of mixing all the data and splitting them into two randomly selected disjoint data sets (i.e., training/testing sets), thus disregarding apps' historical location in the Android timeline, undermines *historical coherence* and yields *significantly biased* and *historically incoherent* results [2, 82]. For instance, this happens when the *testing* set contains past data with respect to the *training* set or when the *training* set may contain *future* data also represented in the *testing* set. Only a very limited quantity of studies in the relevant literature have considered the usage of distinct and *historically coherent* snapshots of Android history for the training/testing split. However, they show significant time gaps between them or other temporal *inconsistencies* (e.g., *malware* and *benign* samples do not belong to the same historical time frame).

Consequently, even though most of the proposed detection systems in the literature support their detection methods with great performance metrics, the aforementioned issues pose serious doubts about the *generalization* capabilities and effectiveness of these solutions to detect evolved and *recent* malware for a long time. In conclusion, the majority of the literature regarding Android malware detection neglect concept drift and its degenerative impact on the proposed detection models.

The impact of the *time* variable on data and malware family evolution has been ignored in the vast majority of Android malware research studies. Just a few studies dealing with Android malware detection have considered the concept drift issue and proposed machine-learning solutions that *adapt* to changes in the data and are able to minimize its detrimental effect over time. In this regard, even though several general approaches have been proposed to detect data *drift* [58, 16, 82], all the specific proposed solutions dealt with API calls [78, 80, 22, 108, 112, 65, 21], an inherently static feature that can also be acquired dynamically. None of the concept drift-related studies considered dynamic features such as system calls, which may enable the induction of more robust systems, as they are relatively immune to obfuscation and encryption techniques that can more easily bypass static API-based detection systems.

A direct consequence of neglecting concept drift is the absence of timestamped data sets available for Android malware research. None of the available data sets in the research domain provide information about the specific historical context to which the samples belong, which might be distinct from the one in which they were collected.

Timestamps are the central elements behind concept drift handling and its proper analysis as they enable the temporal placement of the sample, aiming to provide a *reliable* temporal context.

In this regard, even though some concept drift-related studies did not disclose the timestamp approach used [80], two approaches are observed in the literature: the *compilation time* and VirusTotal's *first seen*. The former is an *internal* timestamp that relates the app to the creation or compilation time of the *apk* bundle. Despite being appointed as the most reliable timestamp in the past [82], and used in related research [82, 16, 108, 21] it has become a *non-usable* approach as the majority of the apps released nowadays have this timestamp set at 1980 [27]. Even though internal timestamps could be deemed *accurate*, they are prone to third-party manipulation, which could lead to temporal misplacement. To prevent timestamp *tampering*, robust temporal approaches can be achieved using *external* timestamps. *Virustotal*'s *first seen*, also referred to as *appearance* or *submission time* in the literature, dates the application with the *datetime* it was first received by the VirusTotal scanning service. This timestamp has been used in the related literature [65, 112, 22] because of its robustness and easy acquisition. However, it is prone to significant delay and temporal misplacement due to the required proactive behavior from the user to generate the *timestamp* for the app (i.e., submission of the file).

Besides, despite the importance of *timestamp* selection to handle concept drift effectively, a related issue in the literature is the absence of research studies exploring the impact of timestamps alternatives or timestamp selection in the effectiveness of the detection models to handle concept drift. The concept drift-related studies in the field did not assess or considered timestamp selection issues nor any alternative to a single timestamp, thus neglecting its impact and essential role in the success of the detection system to adapt to emerging concept drift.

3.2 Explainability: aiming for a better understanding

Explainability or *interpretability* methods have been used to comprehend the decision processes used by the ML-based detection systems on their predictions (i.e., XAI). Understanding the *rationale* behind the model's decisions aims to enhance the trust of experts in the decision systems (i.e., establishing *why* a prediction was made or detecting model *bias*), and meet potential legal requirements regarding algorithmic outcomes for certain use-cases (e.g., medical practice); however, it has also been found useful to improve existing detection systems, for instance, in ransomware detection [86]. The revision of model outputs can help to enhance detection by understanding the underlying processes in the data. Despite the vast body of research on the application of machine learning techniques to Android malware detection, there is an absence of research regarding the usage of explainability methods to assess and *understand* the predictions made by Android malware detection systems [61]. The majority of the focus in the related research concerns performance, disregarding the reasons behind the model's output. Even though some recent studies are found in the field [60, 52], none of them used *XAI* methods to explain, interpret or describe the evolution of mobile malware over time and the related phenomenon, concept drift.

3.3 Consistent cross-device behavior

The usage of *dynamic* features to describe Android apps requires the execution of the sample in a *sandboxed* environment. Considering the plethora of real devices, Android emulators, and operating system versions that coexist at any given moment, locating a *representative* combination of device and OS version to conduct the data acquisition is an arduous task.

System calls are the most commonly used dynamic features for Android malware detection [70]. System calls are the mechanism used by running software to request a service from the OS *kernel*. In Android platforms, system calls enable the collection of the *real* behavior of the running app by capturing the information flow among the OS layers [24]. System calls are an example of *pure* dynamic features, meaning that its acquisition can only be achieved through the execution of the app in a *live* Android environment. For this purpose, *real devices* and *emulators* might be used as acquisition platforms. A *real device* is an actual physical phone running an Android OS version, whereas an *emulator* is a software running on a host computer that simulates almost all the capabilities of a real device [7]. Both types of devices have been employed for collecting data in the related literature, with no one execution platform having a distinctly dominant position. The variety of collection devices observed ranges from the usage of a single [107, 6, 85] or multiple real devices [5, 104, 100] to the exclusive usage of emulators, using either specialized Android sandboxes [31, 51] or general-purpose Android emulators [24, 111, 97, 23, 94, 55]. Even though many different combinations of data sources (i.e., acquisition platforms), dynamic features and algorithms, have yielded significant success in the Android malware discrimination task, the single usage of any of the approaches shows advantages and limitations. They are briefly described as follows:

- Emulators are easy to deploy, manage, and they fit perfectly in automated analysis and detection systems [24], enabling for the mimicking of almost all real device capabilities in a wide variety of *virtual* devices and Android versions without actually having each real device [7]. However, malware with anti-sandbox evasion capabilities can deceive them by not triggering malicious actions if an emulated environment is detected [68]. Although some solutions provide enhancements on this

issue [97], they generally provide limited interaction (i.e., specific triggering events might not be possible such as SMS messages or SIM card detection [31]) and fail to install apps that do not support x86 or x86-64 architecture libraries.

- Real devices are more difficult to manage and integrate into automated systems. For instance, restarting the device to run every sample in a *clean* device can be time-consuming, *rooting* can *brick* the device, and ensure the same exact conditions for all tests might not be possible [67]. However, they provide full interaction with the app, they are inherently immune to anti-sandbox techniques, and show much fewer incompatibility issues.

The numerous acquisition platforms, operating system versions, and configurations described in the related literature demonstrate the presence of an underlying assumption concerning dynamic data. These studies *axiomatically* conceive the dynamic behavior of an app as fully consistent across devices [67] and Android versions [20, 96], even explicitly stated, and, consequently, that the nature of the devices (i.e., emulators or real devices) and the OS version used make no difference to the app's collected behavior. This axiomatic *cross-device behavioral consistency* explains the heterogeneity observed regarding Android platforms and OS versions used in the related research and the lack of common selection criteria in the experimental setups.

However, the results of the studies that have experimented with both kinds of devices challenge the *validity* of the *cross-device behavioral consistency* postulate. For instance, in [4], when *API calls* and *intents*, usually analyzed as static features, were captured dynamically, real devices were found to provide more reliable and stable features for malware detection than emulators, thus leading to a more effective detection outcome.

3.4 Android malware: a complex and ever-evolving threat landscape

The *standard* Android malware detection research study proposes a novel method induced and validated using a *static* and limited snapshot of Android data. If dynamic features are used, they are collected from a single acquisition device. Then the data are mixed and randomly split to train and validate the model. Usually, a high-performance metric is reported (e.g., over 90% accuracy) to support the effectiveness of the method of detecting Android malware and validate the novel work. However, these results only demonstrate the effectiveness of the proposed method to detect Android malware in a *limited* and likely unrepresentative data set. *Static* models could work well if the data features remain constant; however, they are prone to degenerative performance when they are applied to *dynamic* data scenarios. Android data are dynamic, the threat landscape has been consistently changing since the inception of Android and will continue to change due to the constant battle between attackers and defenders, the large attack surface exhibited by these devices, and the constant change of features performed in every *official* API release. Consequently, neglecting *concept drift*, *timestamps*, *historical coherence*, and *cross-device behavior* related issues poses severe concerns regarding the generalization of such proposed methods to *future* and evolved data. This complex, dynamic world cannot be captured properly using static methods, and adaptation is required to achieve long-life effective Android malware detection methods. The exploration of the aforementioned issues constitutes the core part of this dissertation (i.e., Part I, *About Android Malware Detection*).

4 A brief on IoT botnet detection

An IoT botnet is a specific type of computer botnet in which the compromised devices are *Internet of Things* devices, thus presenting analogous schemes and dynamics as computer botnets. In this regard, when a device has its vulnerabilities exploited, thus being compromised, it becomes a *bot*. Bots are grouped in a large community of compromised devices, called a *botnet*. A botnet is typically under the control of a malicious actor, the *botmaster*. The botmaster controls remotely the bots over the Internet, using *command & control* (C&C) servers [93]. This privileged access and control are unauthorized, as there is no consent or awareness from the real owner of the compromised device.

IoT botnets are used to perpetrate a wide range of network-based cyber attacks, from massive SPAM and *phishing* campaigns to distributed denial-of-service (DDoS), the most common use case of botnets. A DDoS attack targets the availability of online resources, such as websites or services. It seeks to deplete the resource by saturating the targeted server or network. As a result, the *crashed* machine or network may become unavailable and unresponsive to legitimate users requests for an extended period of time until the incident is resolved or the attack stops [105]. The attack can cause not only significant financial losses to companies and individuals but also severe loss of trust and reputation [101].

Due to the serious effects of botnet attacks, research studies in the field have concentrated on improving intrusion detection capabilities for IoT devices, seeking to overcome the devices' limited hardware and software resources and security-related capabilities [110]. Machine learning-based solutions and methods have been proposed for such a purpose with remarkable success for multiple botnet attacks [76, 92], but with the major focus on the Mirai botnet [74, 25]. Feature selection and dimensionality reduction techniques have also been used to optimize the feature sets, mainly using filter feature selection methods [79, 15].

In consonance with the attack focus by research studies, all the available data sets used to build and test ML-based IoT intrusion detection systems simulate attack scenarios where the malicious label is represented by attack data and the benign label with *normal* IoT traffic. With minor exceptions [64], all the publicly available data sets for IoT botnet research [76, 81, 17, 59] reproduce *Mirai*, the most prominent IoT botnet and perpetrator of record-breaking attacks [8], and its antecessor *BashLite* [73]. The related data sets simulate different attacks that botnets can perform and also the scanning attack for the recruitment of new members, as part of the *post-attack stage*. Besides, the available data sets share additional characteristics such as only deploying a small number of IoT devices, either real or emulated, in a small-sized network.

As a result, the research efforts in the IoT botnet detection field have focused on the attack and post-attack phases for well-known IoT malware (e.g., *Mirai* and its variants [63]). Therefore, the early detection of the threat, that is, the detection of botnet components at the early stages of botnet deployment (i.e., initial infection and C&C communication) has not been explored in the research or addressed in the available data sets. However, early botnet detection arises as a key element to prevent botnet formation and, consequently, to prevent attacks.

The second part of this dissertation (i.e., Part II, *About IoT Botnet Detection*) focuses on the investigation of aspects aiming to enhance IoT botnet detection at the early stages of its formation, which may help to prevent botnet attacks, and explores the usage of feature selection and interpretation methods to enhance and comprehend relevant aspects related to attack detection.

Part I

About Android Malware Detection

5 KronoDroid: a historical Android data set

This chapter introduces *KronoDroid*, a novel, timestamped data set that provides labeled Android samples encompassing the Android historical timeline from 2008 to 2020, thus enabling the study of concept drift and cross-device detection issues. This chapter also summarizes the main findings of the studies that lead and determined the workflow and characteristics of the *KronoDroid* data set.

5.1 The path leading to KronoDroid

The vast majority of Android malware detection-related research has been optimized for *MalGenome* and *Drebin* data sets. These data sets, collected before 2013, are used as the primary malware source of malware samples in recent studies, a decade after their generation. These widely used data sets are representative of a limited time frame of the Android historical timeline; however, they are not representative of the present malware threat landscape for Android users. The Android malware landscape is characterized by constant evolution, which is the outcome of the constant battle between attackers and defenders in the cyberspace. As a result, a solution tailored for a specific time frame may not be able to generalize well to *posterior* data where the relevant features for effective malware detection may have changed, a phenomenon called *concept drift*. Therefore, an effective solution for Android malware detection should be to detect and adapt over time, modeling and reacting to the changes in the threat landscape. If this dynamism is neglected, the effects of concept drift result in the detection model becoming obsolete due to degenerative performance over time.

Given the ever-evolving nature of the phenomenon, the actual effectiveness of a proposed detection system must be tested in scenarios where the solution faces realistic challenges such as concept drift and data imbalance issues. At the time of this research, none of the existing data sets were representative of the dynamic threat landscape of Android malware in a relatively wide time frame, being limited to reduced snapshots within the extended Android history, i.e., 2008–2022. Consequently, to address and study concept drift issues related to Android malware detection, the generation of a data set encompassing the widest possible time frame of the Android historical timeline was required.

The initial stage of a data set generation involves the exploration of methodological and feasibility issues. In this regard, Publications I to III cover methodological nuances and feasibility approaches related to dynamic and static data collection from Android devices. The related findings were applied in the data collection phase of the data set generation, which is detailed in Publication IV, and materialized in the *KronoDroid* data set.

Publication I is the seed of the posterior work addressed in this dissertation, providing initial insights about the existence of concept drift and other challenges for effective and sustained Android malware detection over time. In this seminal study, the focus was on the analysis of relevant dynamic (i.e., system calls) and static (i.e., permissions) features on two distinct historical time frames of Android malware history. Malware data belonging to different Android time frames (i.e., 2010–2012 and 2016–2018, named as old and new malware respectively) and benign data, belonging to the 2016–2018 time frame, were used. As a result, two distinct data sets were used combining the old malware data and benign data (i.e., L/O), and the new malware data with the benign data (i.e., L/N). The same features were used to characterize each data set. The system calls feature set was composed of 212 features with numeric values reflecting the absolute frequency of each feature for the first 2,000 system calls invoked by the app. The permissions feature set was composed of 147 features containing categorical values (i.e., binary) that indicate the

presence of the permissions for the specific app.

Before inducing machine learning-based classification models, a two-step feature selection procedure was applied to select and rank the relevant features for each data set. In the first step, statistical hypothesis testing was applied to select the features that significantly differ between the classes for each data set. Welch's Test was used for the numeric features and χ^2 for the categorical features with statistical significance level $\alpha = 0.05$. In the second step, *Fisher's score* and *Gini Index* values computed for the numeric and categorical features, respectively, were used to rank the selected features according to their discriminatory power. The feature rankings are provided in Table 2 and Table 3.

Table 2: System calls ranked by Fisher's score [46]

System call name	L/O	L/N
clock_gettime	0.84	1.11
munmap	0.75	0.57
readlinkat	0.69	0.59
connect	0.67	0.52
mmap2	0.63	0.47
prctl	0.61	0.53
madvise	0.54	0.48
ppoll	0.31	0.25
sigaction	0.29	0.30
sigaltstack	0.23	0.21
openat	0.22	0.16
mprotect	0.15<	0.19
futex	0.30	0.15<
rt_sigprocmask	0.24	0.15<
epoll_create1	0.23	0.15<
eventfd2	0.22	0.15<
getppid	0.22	0.15<
clone	0.21	0.15<
sendto	0.19	0.15<
recvfrom	0.18	0.15<
close	0.17	0.15<
getdents64	0.15	0.15<

Table 3: Permissions ranked by Gini index [46]

Permission name	L/O	L/N
ACCESS_NETWORK_STATE	0.46	0.41
WAKE_LOCK	0.45	0.39
INSTALL_PACKAGES	0.42	0.41
READ_PHONE_STATE	0.32	0.45
GET_ACCOUNTS	>0.47	0.47
SYSTEM_ALERT_WINDOW	>0.47	0.46
GET_TASKS	>0.47	0.45
MOUNT_UNMOUNT_FILESYSTEMS	>0.47	0.44
VIBRATE	>0.47	0.44
ACCESS_FINE_LOCATION	0.47	>0.47
BIND_REMOTEVIEWS	0.47	>0.47
USE_FINGERPRINT	0.47	>0.47
CAMERA	0.47	>0.47
BLUETOOTH	0.46	>0.47
READ_LOGS	0.44	>0.47
SEND_SMS	0.43	>0.47
READ_CONTACTS	0.43	>0.47
READ_EXTERNAL_STORAGE	0.33	>0.47

The results showed that the relevant subset of features differed significantly between the data sets, and, more importantly, in the ranking according to discriminatory power. In this regard, the ranked features showed significant deviations between the studied time frames with the only exception of the most discriminatory feature, which was common for both feature sets (i.e., *clock_gettime*). System calls results revealed that, over time, the behavior of legitimate and malicious apps became more similar, with new characteristics becoming more crucial and others losing their discriminatory abilities. An analogous but greater shift was observed for permissions, implying the development of a new character in the data. The classification results, using subsets of common features between time frames, indicated that the observed changes in discriminatory power of features led to changes in prediction performance, suggesting the existence of concept drift and motivating further exploration.

The behavioral data used in Publication I was collected from an emulated device. Even though Android emulators are capable of simulating most of the capabilities of real Android devices, they suffer from limitations at the software and hardware level, which may have led to the observed differences in the case of the system calls feature set. Besides, some of the Android malware included in the experimentation could have had *anti-sandbox* capabilities which may have been able to detect the emulated environment and hide or inactivate the malicious behavior. Therefore, Publication II explored the impact of the collection platform on the acquired behavior from the apps and its implications for cross-device malware detection performance.

Publication II focuses on the comparison of the acquired behavioral data from dis-

tinct device types (i.e., emulators and real devices). For this purpose, an emulator and a real Android device were used, running the same Android OS version and mimicking the internal configurations of both devices to the greatest possible extent. The same data set, composed of 220 Android applications (i.e., 110 malware and 110 benign), was executed on both collection platforms for 60 seconds. The rational behind this specific time-constraint is based on the findings of Publication III. The behavioral profile (i.e., invoked system calls trace) during the execution time of each Android app on both platforms was logged and further analyzed. Table 4 provides descriptive statistics (i.e., mean, median, standard deviation, range, and interquartile range) calculated for the acquired data per class and device type. The statistics revealed inconsistent behavior across classes and device types, as can be observed. In general, the real device data showed more dispersion than the emulator data (i.e., a greater standard deviation and range). More importantly, for the real device, a consistent invocation of system calls not defined in the standard C library for the Linux kernel (i.e., *Bionic* library) used in Android OS was observed. These system calls were anecdotic in the emulator.

Table 4: Descriptive statistics of the acquired data [39]

	Real Device		Emulator	
	Benign	Malware	Benign	Malware
$\bar{\chi}_b$	12,993	11,610	12,601	12,010
s_b	30,357	29,496	15,899	19,853
range	[36, 289,715]	[368, 265,746]	[1,121, 106,076]	[28, 101,867]
$\tilde{x}_b$	6,213	4,033	7,561	4,343
IQR_b	9,245	5,300	9,632	6,174
$\bar{\chi}_{nb}$	1,469	1,367	0	1
s_{nb}	3,641	3,575	1	2
range	[7, 34,281]	[24, 33,032]	[0, 11]	[0, 11]
$\tilde{x}_{nb}$	618	410	0	0
IQR_{nb}	951	532	0	0

To further investigate these initial differences, the exploration focused on the *Bionic* system calls. In this regard, Pearson's linear correlation coefficient (ρ) and Fisher's score were calculated feature-wise per data set to analyze statistical correlation and select the most discriminatory features, respectively. The correlation results show that $\approx 63\%$ of the features in the emulator were highly correlated with some other feature (i.e., $|\rho| \geq 0.80$), whereas in the case of the real device, $\approx 46\%$ of the features showed this characteristic. Fisher's score was used to rank the features according to their discriminatory power. In general, Fisher's scores for the real device were lower than for the emulator, and with completely different orderings. These results enabled us to confirm that the behavior of the same set of apps on different Android devices is not consistent. Additional support for these findings was provided by the binary cross-data set classification models induced, which showed that emulator data could be discriminated relatively accurately by models trained on the real device data but not otherwise, and the fact that a multi-class classification model could be trained to effectively discriminate any class from the generated data sets, i.e., 86% accuracy. This latter fact emphasizes the possibility of building a classifier capable of accurately predicting the class and device of an application based just on the 1-minute data behavioral profile.

As aforementioned, Publication III provided the time-constraints applied in Publication II and Publication IV. More specifically, Publication III performs an analysis of the impact of collection windows for effective Android malware discrimination. Using the same devices as in Publication II, but with an extended data set composed of 330 samples split into 110 samples among old malware, new malware and benign data, each application

was executed and let run freely (i.e., no user interaction) for 15 minutes. This simulates a realistic scenario where an antivirus solution performs on-device collection for posterior class prediction. After each app was processed, the 15-minute-long system call trace was further analyzed. The syscalls traces were analyzed from two perspectives, namely *time-specific system call frequency analysis* and *time-cumulative discriminatory power feature analysis*. In the former, a minute-based histogram is generated per application, reflecting the number of system calls invoked per time-unit (i.e., minute). In the latter, Fisher's score was calculated feature-wise per cumulative time-unit, providing a measure of the discriminatory power evolution of the feature over time.

The application-wise analysis evidenced that, in both Android platforms, the majority of the apps invoked the largest proportion of the system calls of the total collection time during the first minute, referenced as a *1st-minute spike* in Publication III. More precisely, over 88% of the applications analyzed issued the maximum number of system calls in the first minute, invoking less in the subsequent minutes, or remaining inactive. Consequently, from this perspective, the first minute after the boot-up was consistently the most productive in terms of system call invocation across classes and devices.

The feature-wise discriminatory power evolution analysis used Fisher's score as a measure of discriminatory power, thus requiring positive and negative class data. Therefore, two data sets were generated merging the benign data (L) with the old (O) malware and the new (N) malware data, referenced as L/O and L/N, respectively, and for each device (i.e., emulator and real device), resulting in four possible combinations. The cumulative analysis refers to the fact that the data used to calculate the Fisher's score value for the *i*-th minute contains all data up to that specific minute, as opposed to the previous analysis that only included the data issued within the specific minute. The results, provided in Figure 3 for distinct Random Forest models induced using distinct feature sets, showed that in most cases and disregarding the device type, the discriminatory power of the features decreases over time (i.e., lower Fisher's score value), achieving the global *maxima* in the first minute and decreasing *almost* monotonically for the subsequent time frames. In the cases where the decrease was not monotonic, lower peaks or a relatively flat line graph was observed.

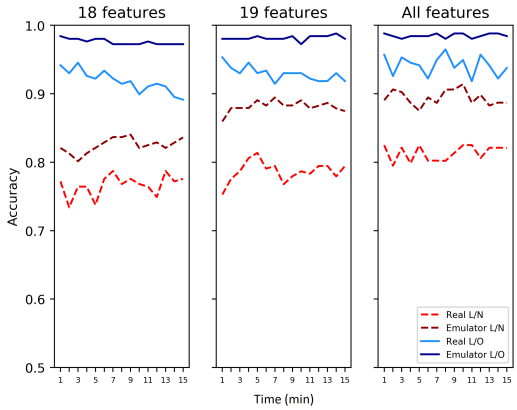


Figure 3: Random Forest models' accuracy [47]

As a result, both analytical approaches converged towards the selection of the first-minute time frame as the optimal time frame to obtain an effective and efficient trade-off between data quantity and discriminatory power. The classification models induced

provided additional support showing that, in general, the accuracy obtained using the first-minute data provided better or similar accuracy than longer time frames.

The main methodological findings of Publication I (i.e., concept drift), Publication II (i.e., behavioral differences across devices), and Publication III (i.e., optimal collection time window), were used as the basis to perform a large data collection effort which materialization is the *KronoDroid* data set, detailed in Publication IV. At the time of this research, the *KronoDroid* data set is the only Android malware data set conceived with a concept drift mindset and suitable for Android malware concept drift exploration. Timestamps are the central element in concept drift analysis as they provide a notion of the temporal context of the application within the historical timeline. In this regard, the inclusion of timestamps as features to describe the data set samples is a distinctive characteristic of the *KronoDroid* data set as timestamp features are not included in any other publicly available Android data set.

5.2 KronoDroid: time-based Android data set

This subsection explains the methodology used to acquire and generate *KronoDroid* data set, and its main characteristics in the form of statistical analysis.

5.2.1 Data set generation

The central concept introduced in the KronoDroid data set is the inclusion of the temporal dimension as a sample descriptor. The naming of the data set emphasizes this focus by referencing *Chronos*, the god of time in Greek mythology. The spelling as *Kronos* was preferred as a tribute to Estonia, the physical location where the data set was conceived, and attending to the particularities of the Estonian language.

KronoDroid data set is the cornerstone of this dissertation as it is the catalyst of the subsequent research performed in the field of Android malware detection contained in this thesis, i.e., from Publication IV to Publication X, and the upshot of the previously detailed studies, i.e., Publication I, Publication II, and Publication III. This relation is conveyed graphically by the diagram depicted in Figure 4.

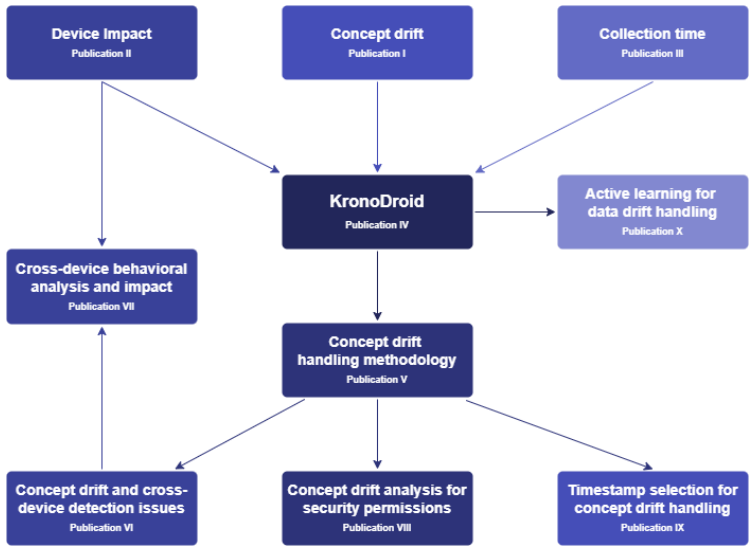


Figure 4: Graphical depiction of the relation among the publications (i.e., Publications I-X)

KronoDroid data collection methodology includes singular aspects to overcome the limitations of the available data sets for Android malware detection, which do not consider the *time* dimension into the data collection loop nor the behavioral differences caused by the collection platforms in the case of dynamic data. The workflow of the data collection process is summarized in Figure 5.

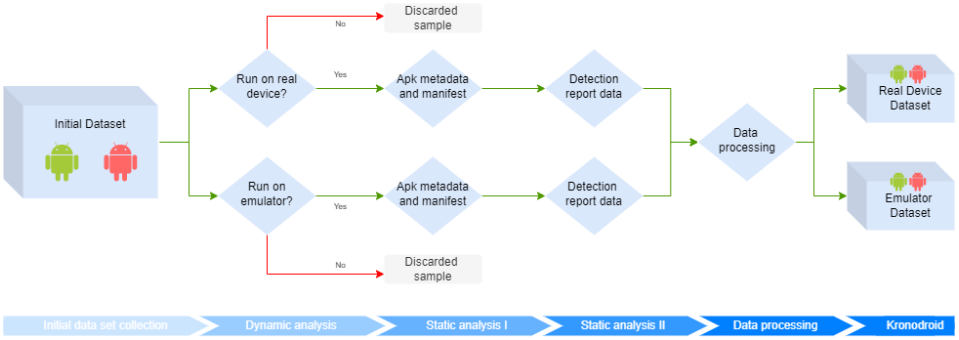


Figure 5: KronoDroid *generation workflow [41]*

The full workflow was composed of 5 sequential stages that run in parallel for both Android collection platforms (i.e., emulator and real device). They are explained as follows:

1. **Initial data set collection**: A total of 93,894 Android app samples were collected from various malware sources and repositories. As depicted in Figure 6, the overlap of the time frames of each data sample according to the specific data source enabled us to encompass the whole Android historical timeline at the time of the research, i.e., 2008–2020. In Figure 6, the length of the boxes indicates the temporal range encompassed by the data gathered from each specific data source while the source data set name and the number of samples acquired are indicated inside the box. The red and green colors indicate the class of the samples, i.e., green for benign apps and red for malware apps. As reported in Figure 6, the malware data set was acquired from four distinct sources [10, 12, 98, 99], totaling 54,834 data samples. The benign data set was collected from three main sources [9, 28, 68], summing up to 39,060 Android apps.

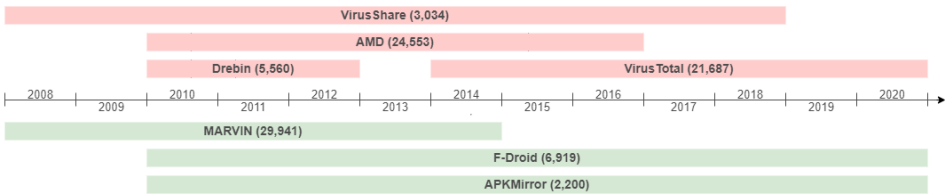


Figure 6: Initial data set timeline and class composition

2. **Dynamic analysis**: Every application within the initial data set was attempted to be installed and executed in both a real and an emulated Android device. Upon successful installation and execution, apps' system call traces were acquired for a 1-minute run-time suggested as the optimal time frame for data collection in Publication III. This procedure was automated using a *bash* script, leveraging *Android Debug Bridge* (ADB) for computer-device communication and *strace* debugging tool

for behavioral trace retrieval, attaching the tool to the app's main process. *Monkey* tool was used to boot-up the application, and no further interaction was performed. As a result, for each device and application within the data set a behavioral profile based on the system calls trace was generated. It was not feasible to install as many apps in the emulator as on the real device due to some applications' intrinsic incompatibility with particular hardware architectures. Consequently, two data sets were generated containing the dynamic profile of each app on each device. In order to ensure the same conditions on both devices, the same Android OS version (i.e., Android 8.0 *Oreo*) and user settings were implemented.

As can be observed in Figure 5, if an application was not successfully installed and executed, it was discarded and was not further processed for the specific device-related workflow. This requirement ensured the collection of dynamic data for each included sample and that the final data sets provided hybrid data for the included data samples (i.e., system calls and relevant *static* data collected in posterior steps as app's descriptors). Therefore, this step acted as a *filter* stage, determining the applications from the initial data set that composed each of the final device-related data sets. The composition of the final data sets is summarized in Table 5.

Table 5: Initial and final data sets class composition [41]

Class	Initial	Emulator	Real Device
Malware	54,834	28,745	41,382
Benign	39,060	35,246	36,755
Total	93,894	63,991	78,137

As can be observed, after the removal of duplicated samples (performed in the data processing stage), 63,991 samples were included in the emulator data set, and 78,137 in the real device data set, which corresponds to 68.2% and 83.2% of the initial data set samples, respectively. Most of the apps that failed to run in the emulator were due to *incompatibility* issues upon installation (i.e., non-compatible architecture) as the emulator inherits the architecture of the host device (i.e., x86) thus causing issues for ARM-specific samples. This fact evidences that due to these incompatibilities, mobile malware detection and analysis can be more challenging in *virtual* devices.

3. **Static analysis I**: Every application that was successfully executed in either of the collection devices, was further processed and relevant *static* features were acquired in two different stages. In the first step, *static* data was extracted from the *apk* archive and the *AndroidManifest.xml* file involving the usage of data extraction tools such as *Androguard*, *Android Assest Packaging tool (aapt)* and *ExifTool*. From the apk archive, metadata such as internal timestamps, filesize, and SHA-256 hash were retrieved. Security permissions, intent filters, hardware features requested and other relevant static data were extracted from the Android manifest.

4. **Static analysis II**: After the extraction of the internal data from the app container, the file was submitted to *VirusTotal* antivirus engine for scanning. A detection report was received for each sample from the scanning service. This stage enabled us to acquire other relevant static data such as detection-related (i.e., external) timestamps, and the verification of the class label (i.e., whether the sample was detected as malware or benign) and extraction of the malware family in the case of malware samples.

5. **Data Processing**: All the data gathered in the previous stages were processed, and data features were manually engineered from the raw files and logs. A total of 489 features were crafted as descriptors or attributes for every data sample on both data sets. Of the data descriptors, 289 refer to dynamic data (i.e., system calls), and 200 to static properties such as security permissions (i.e., 166 features), timestamps (i.e., 4 features) and class labels. Redundant data samples were identified based on hash value and removed keeping just one randomly selected instance.

5.2.2 Data set analysis and main results

Due to compatibility issues that may hinder or impede the malware analysis task (e.g., not including libraries supporting the architecture or the OS version requirements), the data collection process for the data set showed that malware detection and analysis is a more difficult task when using dynamic features in *virtual* environments than in real device platforms. As a result, the emulator data set size is smaller than the real device data set, especially in the case of malware.

Initially the samples were labeled according to the data source. Thus, if a sample belonged to a malware repository, it was labeled as malware. However, when this class label was contrasted with the detection report results, inconsistencies were found. For instance, several benign samples were detected as malware by the scanning service, and a small number of malware apps were not detected by any AV or a significant proportion of them. This generated the suspicion of data *misclassification* issues. As a result, two labels were generated, namely, *soft* and *hard* label. The *soft* label reflects whether the sample was acquired from a benign or a malware data source. Therefore, the possible *soft* label values are contained in the set {0, 1}. The *hard* label applies a set of additional conditions to impute the class label which aims to provide additional evidence or certainty towards the samples class. In this case, the possible labels are {-1, 0, 1} which are defined applying the following rules:

- 0 or *benign* class: refers to a sample with a zero-valued malware detection ratio *and* it belongs to a benign data source.
- 1 or *malware* class: refers to a sample with a non-zero malware detection ratio *and* it belongs to a malware repository *and* a malware family is identified for the sample.
- -1 or *indefinite* class: any mismatch with the two previous conditions. It indicates that inconsistencies are identified and that further inspection is needed to prevent misclassification issues.

Based on this categorization, the following data sets are defined:

Table 6: Data sets class label composition [41]

Class	Emulator			Real device		
	Soft label	Hard label		Soft label	Hard label	
		Indefinite	Definite		Indefinite	Definite
Malware	28,745	91	28,654	41,382	165	41,217
Benign	35,246	4,437	30,809	36,755	4,856	31,899
Total	63,991	4,528	59,463	78,137	5,021	73,116

As can be observed in Table 6, the indefinite category found more inconsistencies for the benign data. However, they might be false positives due to the strict hard labeling rules implemented. For example, the *soft* labeling approach would classify a benign sample as benign even if an AV detected it as malware, whereas the *hard* labeling rules would classify

it as indefinite. The use of either label may result in effective malware detection systems, and according to the analysis, the majority of the indefinite class may correspond to *false positives*. However, for tasks where label certainty is essential, the *hard* label offers an additional degree of trust certainty towards the sample class. The accurate analysis of concept drift might be one of such tasks.

Besides the importance of class certainty, a central element in concept drift analysis is the timestamp used to locate the data samples within the historical timeline. In this regard, there is no single timestamp approach deemed as completely accurate and reliable in all cases. Thus, distinct timestamps may provide distinct historical locations of the apps within the Android timeline, which, in turn, could provide different concept drift modeling. In general, the more *accurate* the timestamp, the better the concept drift model and the more precise the data evolution analysis. For this purpose, timestamp acquisition was included in the data collection workflow. Kronodroid provides four possible timestamps per sample: two *internal* timestamps and two *external* timestamps. An *internal* timestamp is retrieved from the internal files of the application, whereas an *external* timestamp is determined by a third-party based on different parameters. The *earliest modification* and the *last modification* are the internal timestamps provided by *Kronodroid*, which correspond with the earliest and last modification *datetime* retrieved from any file inside the apk archive, respectively. The external timestamps are based on *VirusTotal*'s detection report, defined as *first seen* and *first seen in the wild*. The former provides the date and time of the first submission of the file to the scanning service, whereas the latter, reports the first time the application was seen anywhere on the Internet. Either of the approaches shows advantages and disadvantages. The internal timestamps are prone to manipulation by malicious actors; thus deliberate manipulation can cause data misplacement. Nevertheless, when not tampered with, they may locate the sample more accurately than the external timestamps, especially the *last modification* timestamp. The external timestamps cannot be manipulated by attackers; however, due to their *proactive* nature (i.e., depending on users' submission time) they are prone to delay and generate temporal displacement. A more detailed inspection of the distinct timestamp approaches and their suitability for concept drift purposes is provided in Publication IX.

The hard labeling rules required that to impute a sample as malware, a malware family attribution could be retrieved from the detection report. Malware family attribution is a controversial issue in the cybersecurity domain. Despite the existence of well-known malware families and a myriad of malware variants, there is no consensus nor naming convention on malware family denomination and identification across AV vendors or malware analysts. For instance, a sample detected as malware by the scanning engine was named in eight different ways, including vendor-specific cryptic denominations and well-known names. Therefore, family name attribution becomes a challenging task in a large data collection study. In order to provide a malware family for each *KronoDroid* malware sample, a heuristic approach was implemented. Firstly, a database of malware family names was generated from malware databases and research studies. When different denominations for the same family existed, they were abstracted into the same name separated by "/" (i.e., *Airpush/StopSMS* family). Secondly, all the detection reports (i.e., including benign samples) were parsed and all the positive scanner results, which might identify the sample with a family name, were compared with the abstractions in the database. After this, a family name from the database was imputed to every sample based on the majority of the vote towards a family name from all the positive detection outputs. In the case of a positive detection but no malware family was imputed (e.g., the malware family was unknown or it was not included in the database), the sample's report was manually in-

Table 7: Top-15 malware families in the final data sets [41]

Emulator			Real Device		
Family	Total	%	Family	Total	%
Airpush	6,521	27	Airpush	7,775	22
Boxer	3,557	15	SMSreg	5,019	15
Malap	2,574	11	Malap	4,055	12
FakeInst	2,158	9	Boxer	3,597	10
Agent	1,837	8	Agent	2,934	9
SLocker	1,822	8	FakeInst	2,384	7
BankBot	1,241	5	SLocker	1,846	5
FakeApp	1,064	4	BankBot	1,297	4
Dowgin	772	3	Dowgin	1,145	3
GinMaster	595	2	FakeApp	994	3
Kuguo	513	2	DroidKungFu	990	3
SMSreg	497	2	Kuguo	843	2
Youmi	447	2	GinMaster	827	2
DroidKungFu	269	1	Youmi	628	2
Simhosy	232	1	Simhosy	399	1
Total	24,099	100	Total	34,733	100

spected and a family name was imputed (if applicable). The new malware family name was included in the database and the process was repeated for all the samples aiming for results consistency.

This heuristic procedure yielded a malware family imputation for 99.7% of the malware samples within the data sets. This translates into 209 different malware families included in the emulator data set and 240 in the real device data set. The difference may be expected as a result of the significantly distinct number of malware samples on both data sets, as shown in Table 5. However, it evidences that certain malware families might be specially tailored for ARM-based systems (i.e., real devices) and incapable of running or being analyzed in x86-based emulators. The inadequacy of emulators to deal with those specific malware families limits the capabilities of Android emulators to perform forensics analysis and be used as reliable detection platforms.

Despite the different number of malware families and samples, the top-15 malware families are, with varying proportions, the same for both data sets, grouping approximately 84% of the total malware samples. They are summarized in Table 7. Furthermore, these most prevalent malware families can be embedded into four major malware categories: *adware*, *fraudware*, *spyware* and *ransomware*. The colors of the table reflect this higher level of abstraction, indicating the degree of threat they pose for the end-user. Adware trojans (i.e., Airpush, Agent, FakeApp, Kuguo, Dowgin and Youmi) are the most prevalent malware families in the data sets, and they are indicated with the lightest color in the table. The greatest the color intensity, the more dangerous the threat. Thus corresponding in increasing order to fraudware samples (i.e., Boxer, FakeInst, and SMSReg), ransomware (i.e., SLocker), and spyware families (i.e., DroidKungFu, GinMaster, BankBot, Simhosy, and Malap).

The usage of timestamps to locate malware families along the Android historical timeline provides different distributions and data trends that can help to explore malware outbreaks and assess to some extent the reliability of timestamps. The combination of temporal aspects with other data attributes such as malware family distribution is sketched in Publication IV, and further explored in Publication VIII and Publication IX.

The dynamic data collected from malware and benign samples evidence different behavioral profiles on each platform. The descriptive statistics computed and provided in Table 8 emphasize the existence of behavioral differences across Android platforms.

Table 8: Descriptive statistics of system calls [41]

Device	Class	Label	Statistic			
			Mean	Median	Std. dev.	Sample size
Emulator	Malware	*Soft*	7,711	2,363	27,893	28,745
		Hard	7,694	2,361	27,911	28,654
	Benign	*Soft*	8,755	2,959	20,995	35,246
		Hard	8,664	2,890	20,771	30,809
Real Device	Malware	*Soft*	10,390	3,258	26,232	41,382
		Hard	10,410	3,267	26,272	41,217
	Benign	*Soft*	2,878	1,148	13,386	36,755
		Hard	2,792	1,134	12,860	31,898

Table 9: System calls sets and usage statistics [41]

Device	Class	Label	Syscalls set (%)	Most used syscalls	Most issued syscalls	Total syscalls
Emulator	Malware	*Soft*	34.4	ioctl, getuid32,	read, getuid32, write,	221,659,298
		Hard	34.4	mmap2, futex, close	epoll_pwait, ioctl	220,454,507
	Benign	*Soft*	42.4	ioctl, getuid32, mmap2,	read, write, ioctl,	308,595,229
		Hard	41.7	close, futex	recvfrom, epoll_pwait	266,918,501
Real Device	Malware	*Soft*	39.2	clock_gettime, getuid32,	clock_gettime, ioctl,	429,999,343
		Hard	38.9	ioctl, futex, mmap2	getuid32, mprotect, SYS_329	429,069,377
	Benign	*Soft*	44.4	clock_gettime, getuid32,	clock_gettime, getuid32,	105,779,886
		Hard	43.8	ioctl, writev, read	ioctl, SYS_329, mprotect	89,051,861

In this regard, even though the data set size might be different for malware samples, the results show inconsistencies in the behavioral averages for the classes between devices. More interestingly, this is especially significant in the case of benign samples, which show significantly different statistics across devices with similar data set sizes. In general, the results show that benign apps invoked more system calls in the emulator environment than in the real device, whereas the opposite is true for malware. These differences are further emphasized when the other statistical figures are compared. Table 9 provides four additional comparative items for a thorough behavioral comparison. The *syscalls set (%)* column provides the proportion of system calls from the system calls set that were used by the apps in that specific data set at least once. The *most used syscalls* and *most issued syscalls* provide the top-5 syscall set used by the applications in the specific data set, and the top-5 of most issued system calls concerning the total number of system calls issued on each data set, respectively. Lastly, the *total syscalls* column provides the sum of all individual syscalls invoked by all the applications in each data set.

The *most frequently used* and *issued* system call sets vary between classes within the same device, although mainly between devices, as demonstrated in Table 9. For example, the top five system calls for the emulator do not contain *clock_gettime*, the system call that is issued and utilized the most on the real device. Besides, the figures for total syscalls confirm the greater verbosity of the benign apps in the emulator compared to the real device, even though a similar proportion of syscall features from the whole syscalls set is used. In this regard, it is worth noticing the difference between classes in the syscalls set usage. Benign apps use consistently a wider range of system calls across devices than malware, especially in the emulator case. Even though at first glance may appear that a larger proportion of syscalls are used in the real device compared to the emulator, a detailed analysis suggests the opposite. The whole system calls feature set includes x86-architecture specific syscall features, defined for the emulator and real device (i.e., 212), and additional features just defined for the real device (i.e., 76), thus totaling 288 features. The same feature set was used for both devices for the sake of consistency for the data features. Thus, when the common feature set is considered, the proportion of system calls

Table 10: Descriptive statistics of permissions [41]

Class	Perm	Label	Permissions statistics				Custom usage (%)	Most used permission set (% apps)	Perm set (%)
			Mode	Mean	Median	S.D.			
Malware	Std	*Soft*	12	13.4	11	9.1	≈ 42	Internet (97.7) Read_Phone_State (91.6) Access_Network_State (84.9) Write_External_Storage (81.0) Access_Wifi_State (66.3)	≈ 78
		Hard							
	Custom	*Soft*	1	4.3	2	4.7			
		Hard							
	All	*Soft*	9	15.2	11	12.4			
		Hard							
Benign	Std	*Soft*	2	4.2	3	4.2	≈ 14	Internet (81.7) Access_Network_State (51.3) Write_External_Storage (36.8) Read_Phone_State (27.4) Access_Coarse_Location (20.1)	≈ 87
		Hard		3.9		3.9			
	Custom	*Soft*	1	1.8	1	1.8			
		Hard		1.7		1.4			
	All	*Soft*	2	4.4	3	4.9			
		Hard		4.2		4.4			

from this restricted feature set (i.e., 212) used by the emulator grows to 46.7% and 57.5% for malware and benign data, respectively. As a result, benign apps show a distinct, more active behavior in the emulator than in the real device, a fact that cannot be confirmed for the malware set. These results provide additional support to the behavioral differences spotted in Publication II and Publication III.

The analysis of security *permissions*, the most used static feature set in Android malware studies [70], provided supplemental insights about the differences between benign and malware applications. Publication IV provides a thorough comparison between both data sets despite the fact that static features are *immutable* features of the applications, i.e., they do not change across devices, as opposed to dynamic features. This is because the two data sets are comprised of distinct numbers of samples, which affects the descriptive statistics that summarize them and provide relevant comparative statistics. Notwithstanding that, the main differences between classes can be broadly grasped by focusing on the characteristics observed for the real device data set due to its larger data size. For this reason, it is provided as an illustrative example. Table 10 provides summary statistics for standard, custom and all permissions (i.e., sum of standard and custom) in the first 8 columns. The *custom (%)* column provides the proportion of each class that defined custom permissions. The *most used permission set* is provided in the subsequent column with the percentage of apps that defined each permission in parenthesis. The last column reports the coverage of the whole permissions set per class.

Based on the data reported in Table 10, notable differences can be observed in the usage and definition of permissions between malware and legitimate applications. Malware applications request significantly more permissions and define more custom permissions than benign apps. Furthermore, the definition of custom permissions is three times more for malware apps than benign apps. Although benign apps often request fewer permissions than malicious apps, they cover a larger spectrum of the permissions set. The proportion of applications that requested them is substantially lower for the *innocuous* class, which is consistent with the lower average number of permissions required by this class of apps, despite the fact that the most used permissions sets are comparable. Therefore, in general, malware apps request more permissions than legitimate apps, both standard and custom, thus requesting over-privileges to the user on the system. Besides, malware permissions requests prioritize more than legitimate apps the access to internet connectivity (by any means) and the access to sensitive user data.

The distinctive request in permissions between classes is also confirmed by the frequency distributions of requested permissions depicted in Figure 7. As can be seen in Figure 7, the shape of malware and legitimate distributions are remarkably different re-

garding the number of requested permissions, with a minor overlap between the distributions below 20 requested permissions (i.e., shady area). Based on these graphs, the majority of legitimate apps request less than five permissions, whereas most malware apps request over five permissions.

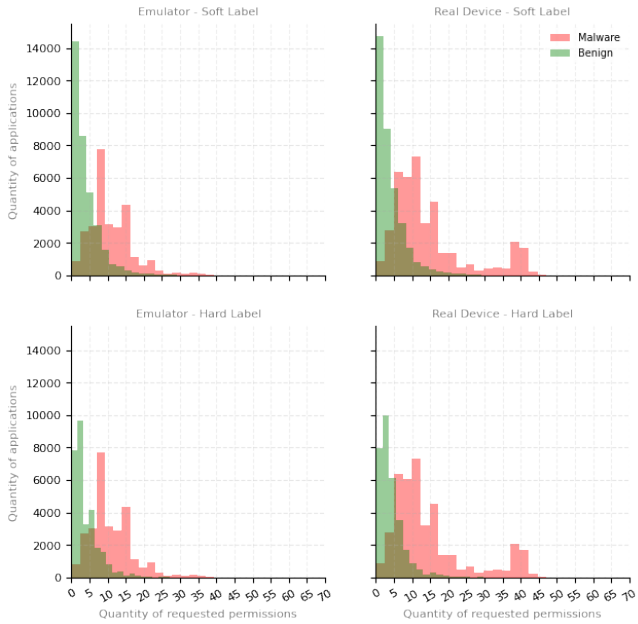


Figure 7: Frequency distributions of requested permissions per class and label [41]

The statistical analysis of *KronoDroid* data has displayed relevant insights concerning Android malware and benign data that can help to build better detection systems using static, dynamic, or hybrid features. However, the distinctive point of the *KronoDroid* data set is its focus on time. *KronoDroid* is the first Android data set that incorporates *timestamps* as features, thus enabling the consideration of data evolution, a variable that has been neglected by the vast majority of Android research studies which are *optimized* for static snapshots of malware data. Even though *KronoDroid* could be utilized for distinct purposes such as family analysis and class-based differential exploration, as shown before, its main aim is to help in Android data concept drift modeling and analysis with the objective of developing more robust and long-term effective detection systems. The data set is the cornerstone of the subsequent research performed and detailed in the next chapters of this dissertation.

5.3 Chapter summary

This chapter presented the seminal works that led to the design and generation of the *KronoDroid* data set. *KronoDroid* is a novel, labeled, timestamped, and hybrid-featured Android data set tailored for Android malware research that enables the study of concept drift, malware family evolution, data imbalance, and cross-device detection challenges. If they are not addressed, these problems will be encountered by detection systems in production environments, decreasing their effectiveness over time and impeding the long-term detection of Android malware. None of the publicly available Android malware data sets enabled the study of such detection challenges prior to *KronoDroid*.

6 Concept drift on behavioral data: detection, handling and characterization

KronoDroid data set enabled us to study the evolution of Android data, and, consequently, concept drift-related issues from different perspectives. The following chapter deals with the emergence of concept drift on behavioral data. It proposes a solution to handle it, which can also be used to characterize it, leading to a better comprehension of the changes. The present chapter focuses on the main contributions and findings of Publication V.

6.1 Workflow overview

This study focused on the analysis of concept drift in Android apps from a dynamic perspective (i.e., system calls). Due to the larger size of *KronoDroid*'s real device sub-data set it was preferred for this exploration, providing more applications than the emulator data set for the same period of time (i.e., 2008–2020). Therefore, *KronoDroid*'s real device dynamic features were utilized (i.e., 288) along with the *hard* label and timestamps. The hard label was preferred as it increases the certainty of the app class, thereby generating further confidence in the concept drift analysis. Of the four timestamps provided by *KronoDroid*, the *last modification* and *first seen* were used. These timestamps were preferred over the other options as they provide wide timeline coverage, reliability, and *relatively* accurate location of the apps within the Android historical timeline. In this regard, a more thorough exploration of these aspects was performed in Publication IX. As a result, 78,137 Android apps described by 288 system call features, the label, and two distinct timestamps were used as input data.

The methodology to explore concept drift-related issues and effective handling was split into three sequential stages, namely, *detection*, *handling* and *characterization*. These consecutive steps enabled us to *demonstrate*, *address* and *visualize* concept drift in behavioral Android data, respectively. The rationale behind the successive steps is that the previous one justifies the following one. For example, only if concept drift exists there is an actual need to address it. Figure 8 details the purpose and flow of these sequential stages which are detailed in the following sections.

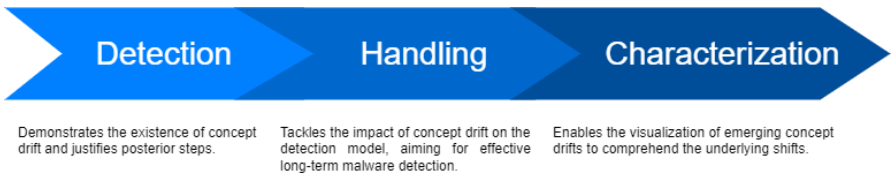


Figure 8: Concept drift detection, handling and characterization workflow

6.2 Concept drift detection

The concept drift detection phase aims to demonstrate the existence of concept drift in Android historical data. This phase was composed of three sequential stages, namely *data pre-processing*, *feature selection*, and *drift detection*. The whole process is schematized in Figure 9 and explicated in the following paragraphs.

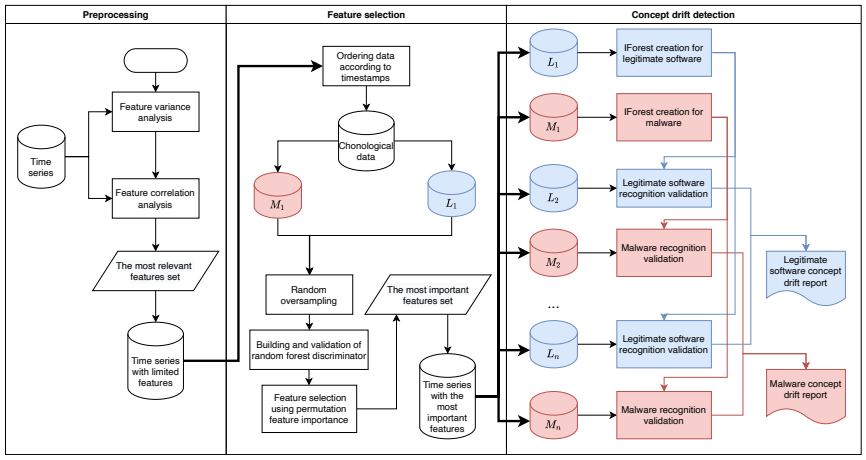


Figure 9: Concept drift detection workflow [42]

6.2.1 Data pre-processing

The initial set of features (i.e., 288) was pre-processed using a sequential procedure that aimed to remove irrelevant and redundant features. The outcome of this stage was a refined set of features obtained after performing the steps described as follows:

1. *Variance analysis*: homogeneous and null-valued features were removed.

2. *Correlation analysis*: Pearson's linear correlation coefficient (r) was calculated pair-wise for all data features. Highly correlated features (i.e., $r > 0.80$) were removed.

3. *Distribution analysis*: the adherence to the *normal* distribution of the remaining features was assessed using statistical tests.

The results of the application of the first two steps resulted in a refined set of 97 features out of the initial set of 288. The statistical normality tests performed confirmed that none of the features were normally distributed, as evidenced by Figure 10, which prevented the application of *parametric* methods for concept drift detection.

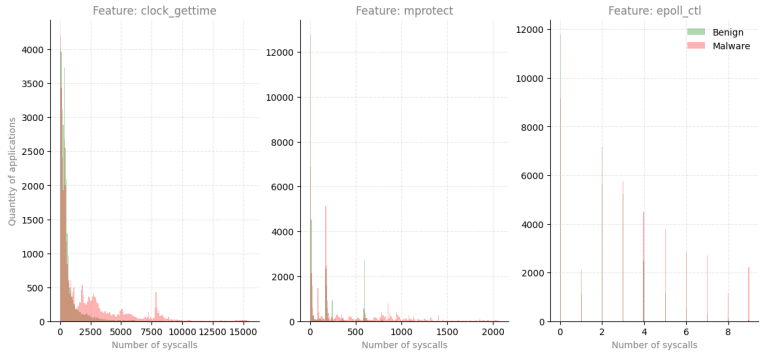


Figure 10: Feature distributions [42]

6.2.2 Feature selection

To assess the existence of concept drift, the data set was ordered chronologically and divided into disjoint consecutive time periods. For n periods, two series of data subsets were generated. In the first series, the set M_i consisted of malware samples labeled with the timestamp (i.e., *last modification* or *first seen*) and located on the i-th period. The set L_i, defined for the second series, was treated analogously and composed just by benign samples. Next, the most important discriminators were selected for the first period (i.e., $i = 1$), taken as the *initial* or *baseline* period (i.e., feature selection). More specifically, the *permutation feature importance* technique (detailed in Section 6.4) was applied to a classifier model induced using $M_1 \cup L_1$, thereby enabling the selection of the most relevant features for this baseline classifier.

As the initial period classification model provided high performance (i.e., over 95% accuracy), the logical derivation is that the selected features were able to recognize classes L_1 and M_1 effectively. A concept drift-related question is whether the same features could be successfully used to recognize classes L_i and M_i for $n \geq i > 1$.

6.2.3 Concept drift detection

To address the previous issue, One-Class Drift Detection models (OCDD) [34] were used to analyze the impact of concept drift in the generated series. Based on the fact that the selected *important* features were used successfully for the classification task on the $M_1 \cup L_1$ data set, one-class anomaly detectors (OCDD) were induced using L_1 and M_1 separately, to assess data drift. This approach enabled us to analyze concept drift for malware and benign data in a more controlled way, eliminating the class relations influence. The induced models were tested separately with the data belonging to the same class in the subsequent time periods (i.e., L_i and M_i sets, where $n \geq i > 1$, described by the selected feature set). The ratio of samples recognized by the initial models for each period was retrieved (i.e., negative detection rate for each class). These ratios enabled us to assess and detect concept drift on the data over time. More specifically, if the *important* features for the first period were not able to describe the modeled class effectively in a posterior time frame, the ratio dropped, thus suggesting data drift. As the ratio of correctly recognized samples declines, the shift impacts the classification results, qualifying the performance drops as concept drift.

6.2.4 Experimental results

Due to data constraints and model-building requirements (i.e., highly accurate models for both timestamps), the initial period selected for concept drift detection was the second semester of 2011. Of the initial set of features (i.e., 97), the feature selection procedure applied to the initial period data set yielded different subsets of *important* features for each timestamp. More specifically, 32 features were found to be *important* for the *last modification* timestamp and 17 for the *first seen* timestamp.

These feature sets were used to build the one-class anomaly detection models. As a result, for each class (i.e., malware and benign) in each timestamp-based data set, a one-class anomaly model was generated, using the corresponding feature set as model features. Then, the malware and benign data belonging to *posterior* time frames were split into six-month periods (i.e., from 2012 to 2020) and used as test sets for the corresponding timestamp-class model. Besides, for each timestamp-class combination, three anomaly models were induced using distinct subsets of features from the *important* feature sets (i.e., best five features, best 10 features, and all features).

The results are provided in Figure 18 where the models' accuracy performance is re-

ported. The line graph on the left shows the results related to the *last modification* timestamp, whereas the line graph on the right provides the data related to the *first seen* timestamp. The six anomaly models generated for each timestamp are reported with different colors and line styles. The color relates to the data class (i.e., red for malware and green for benign apps). The line style informs about the subset of features that was used to build and test each specific anomaly model. The horizontal axis provides the testing period, whereas the vertical axis provides the accuracy value retrieved for each 6-month period. The *.1* value attached to the year number informs about data belonging to the first semester of that year (e.g., 2012.1), whereas *.2* reflects the data located in the second semester (e.g., 2012.2). As a result, six anomaly detection models were built and tested per timestamp (i.e., three per class) encompassing the whole 2012–2020 time frame.

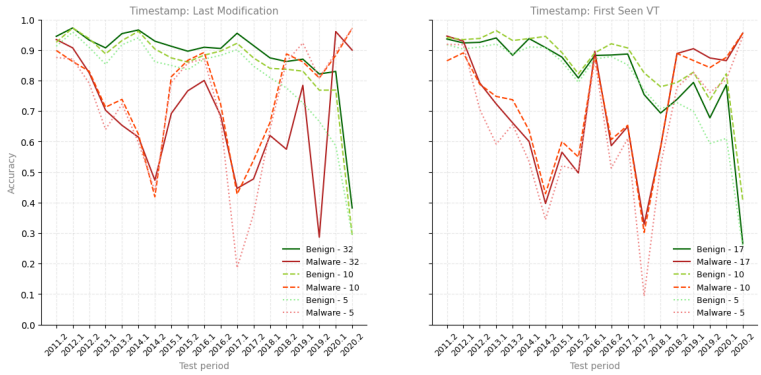


Figure 11: One-class anomaly detection models performance - real device data [42]

The results provided in Figure 11 demonstrate the existence of concept drift in the data. The irregular and overly fluctuating scores prove that the same set of features and values are not useful in all time frames to discriminate either of the classes.

In benign applications, an *incremental drift* dominates. The number of recognized observations slightly goes down over time to dip in the last period in a *sudden drift*. However, the etiology of the last dip might be extraneous due to the scarcity of benign data for that period.

Concept drift is especially evident for malware data. In all cases, the models display remarkably distinct accuracy scores from period to period, indicating concept drift and implying that the significance of the initial features for the classification models changed dramatically. Both timestamps provide a similar scenario, with the initial models performing well on data from closer periods and losing discriminatory power over time. The initial set of important features appears relevant again in 2016.1 and 2019.1, attaining high accuracy scores, but loses significance in the following periods, resulting in data drift and poor discrimination performance. Such behavior is similar to *blips* in the concept drift typology. However, it could be related to a recurrent threat from the initial period, emerging again 2016.1 and 2019.1 periods.

Figure 11 evidences the presence of concept drift in Android historical data, which is especially pronounced in the malware case. The emergence of concept drift in the data requires the implementation of an adaptive detection solution capable of handling it effectively. This issue is addressed next.

6.3 Concept drift handling

To build long-lasting and robust Android malware detection solutions, the detection systems must be capable of adapting and learning from the changes in the data to maintain high and stable performance over time.

6.3.1 The proposed solution

A *data stream* can be defined as *a countably infinite sequence of elements* that become available over time [72]. They are characterized by a cumulative, continuous, rapid, and evolving nature, and pose a variety of challenges such as one-pass constraints, concept drift, resource restrictions, and massive-valued features [1]. Even if it does not present all of these difficulties, Android malware detection may be considered a data stream because it exhibits high data volume, ongoing app release, and constantly changing data. Consequently, Android malware concept drift may be efficiently managed when approached from a data stream viewpoint.

The proposed solution to handle concept drift in Android malware data is a modification of the algorithm proposed by Zyblewski at el. [114] to deal with concept drift issues for imbalanced data streams. The modifications and simplifications performed address Android concept drift particularities, boosting the detection performance on the application domain. Publication V is devoted to the implementation of the proposed solution and provides the experimental results.

A schematic diagram of the solution used to handle Android concept drift is depicted in Figure 12. Based on this diagram, the following paragraphs explain the inner workings of the proposed solution, emphasizing the modifications suggested to handle Android data characteristics.

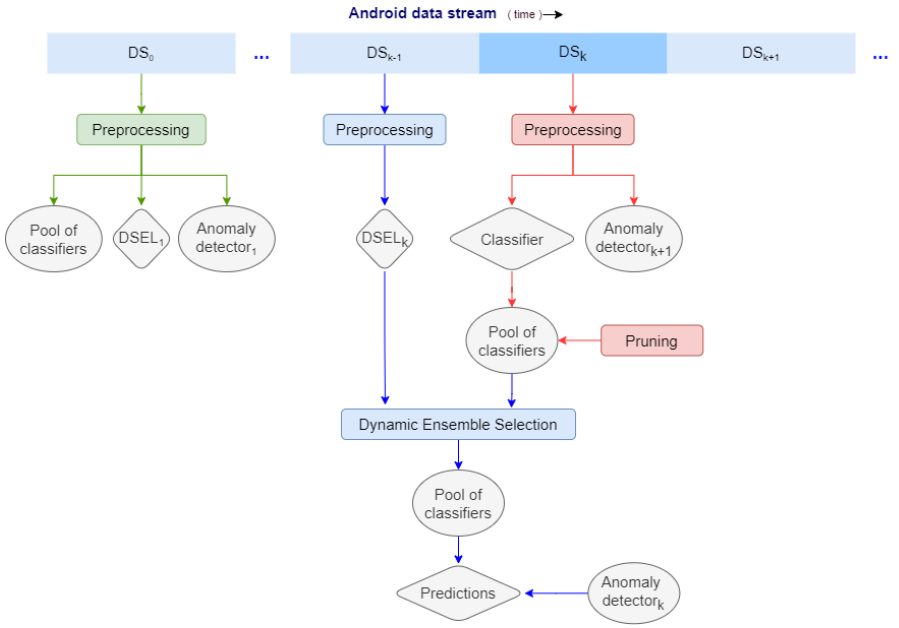


Figure 12: Scheme of the proposed solution for Android concept drift handling

In Figure 12, the green line and related boxes refer to the special treatment of the initial chunk based on the modifications proposed. The red and blue lines and boxes follow the

workflow of the training and prediction phases, respectively, for each subsequent data chunk in the stream of Android data. The training and prediction phases are provided in pseudo-code in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1: Training phase of the proposed framework

Input:
Stream - Data stream
S - Fixed size of the classifier pool
$\Pi \leftarrow \emptyset$ - Pool of classifiers (initially empty)
$\lambda \leftarrow -1$ - Sample size of the anomaly detector
Symbols:
DS_k - Data chunk
Ψ_k - Bagging classifier
L_k - Legitimate data portion of the data chunk
Φ_k - Anomaly detector

```
foreach k, DS in Stream do
    if k == 0 then                                                  // first data chunk
        IDS ← splitInitialDataset(DS_k, |π|)                        // split data chunk
        for i ← 0 to S − 1 do
            Ψ_k ← trainClassifier(IDS_i)                            // train classifier
            Π ← Ψ_k                                                 // add classifier to the pool
        end
        Φ_k ← trainAnomalyDetector(L_k, λ)                          // train anomaly
    else                                                            // rest of the data
        Π ← pruneWorstClassifier(Π)                                 // purge pool
        Ψ_k ← trainClassifier(DS_k)
        Π ← Ψ_k
        Φ_k ← trainAnomalyDetector(L_k, λ)
    end
end
```

Algorithm 2: Prediction phase of the proposed framework

Input:
Stream - Data stream
Π - Pool of classifiers
Φ_k - Anomaly detector
Symbols:
DS_k - Data chunk
$y_{k\,pred}$ - Predicted labels for the samples in the current chunk
Π_{Dk} - Ensemble of classifiers selected using a DES algorithm
$DSEL$ - Dynamic ensemble selection data set

```
foreach k, DS in Stream do
    if k == 0 then                                                  // first data chunk
        DSEL ← preprocess(DS_k)                                     // store DSEL for next step
    else                                                            // rest of data chunks
        Π_Dk ← dynamicSelection(Π, DSEL, DS_k)                      // DES step
        y_kpred ← predict(DS_k, Π_Dk)                               // prediction step
        y_kpred ← anomalyDetector(y_kpred, Φ_k)                     // refinement step
        DSEL ← preprocess(DS_k)
    end
end
```

The proposed solution works as follows: when the first data chunk is received (i.e., $k = 0$), the whole chunk is processed, splitting its *n* elements into S ordered and equal-sized data chunks. Each data subset is used to train a new classifier which is added to the *pool* of classifiers. As a result, a full pool of classifiers is generated after the processing of the first chunk, thus the full pool is available for the testing phase of all the subsequent data chunks. Besides, the set of legitimate samples from the initial data chunk is used to induce an anomaly detection model. The last processing step of the initial chunk ińvolves the storage of the whole initial chunk as the dynamic ensemble selection data set (*DSEL*) for the next chunk. The *DSEL* is used to select the best classifier ensemble from the classifier

pool for each data sample in the new data chunk.

This concludes the processing of the initial data chunk, used for initialization purposes, being the only one with distinct processing steps in our proposed solution. For all subsequent data chunks, the same *testing-then-training* procedure is applied. This procedure is explained as follows.

After the first chunk is processed, when a new data chunk is received, the prediction/testing phase is applied first. Thus, upon the arrival of the new data chunk, a dynamic ensemble selection algorithm is fit with the previously stored *DSEL*, the classifiers pool, and the new data chunk. This step aims to select the best ensemble of classifiers to predict the labels for each sample in the new data chunk. The fitted dynamic ensemble model is used to forecast the class label of the *n* elements of the data chunk, generating the initial set of predictions. These initially assigned labels are then refined, based on custom-generated rules using the forecast of the anomaly detector for each sample. The outcome of this step is the final prediction for all the samples of the new data chunk. The anomaly detector helps to support or challenge the class prediction by the classifier in borderline cases where the anomaly model may provide more reliable results. Finally, the new data chunk is stored as the *DSEL* for the next chunk. This concludes the first processing step of the new data chunk, the prediction phase, detailed in Algorithm 2 and depicted by the red line flow in Figure 12.

The next step for the new chunk is the training phase, described in Algorithm 1.

The training phase uses the whole new data chunk and the outcome of the previous phase to update the pool of classifiers and generate a new anomaly detector. More specifically, the worst-performing classifier on the new data chunk is removed from the classifier pool. Then, a new classifier is trained using the samples from the new chunk and their predicted labels. The new classifier is added to the pool, which is again composed of S classifiers. Removing an *aging* classifier and inserting a *new* classifier keeps the pool at the specified size while updating its capabilities to accurately forecast new data, thus being able to adapt and react to emerging concept drift. The legitimate portion of the new data (i.e., L_k) is used to generate a new anomaly detector that will be used in the predictive step of the next data chunk. This last step concludes the processing of the data chunk.

This *testing-then-training* cycle is repeated for all the subsequent data chunks in the data stream *ad-infinitum*, enabling the system to address concept drift issues effectively and efficiently without needing operational changes in the system.

It is worth noticing that the system is governed by hyper-parameters that have not been discussed. Publication V provides further details and acts as a reference for this discussion. It also provides a detailed explanation of the modifications performed to the original algorithm which can be summarized as:

- The induction of a complete pool of classifiers available from the initialization stage (i.e., chunk 0).
- The addition of a refinement step for the predictions using a supportive anomaly detection model.
- The overall simplification of the stages and removal of special chunk treatments (i.e., with the exception of the initial chunk).

6.3.2 Experimental results

The proposed solution was used to address *KronoDroid* data concept drift. As the data set is not a real data stream and encompasses a long period of time, the timestamps

were used to locate the data within the Android historical timeline and the data was divided into time-constrained consecutive data chunks. Due to data constraints and system hyper-parameters selection, thoroughly explained in Publication V, the data for the experimental setup was partitioned into three-month data chunks encompassing the years from the third quarter of 2011 to the second quarter of 2018. The *last modification* and *first seen* timestamps were used, thus two concept drift-handling detection models were induced.

The F_1 score performance of the system using the *last modification* timestamp is provided in Figure 13, while Figure 14 reports about the results when the *first seen* timestamp was used. In both cases, the performance of the models using the proposed solution are compared with two naive solutions and the original algorithm [114]. The two naive solutions correspond to two different *static* models, one induced using the data from the initial period and another one using the data of the second time period. These static models were never updated and they were tested with all subsequent data chunks.

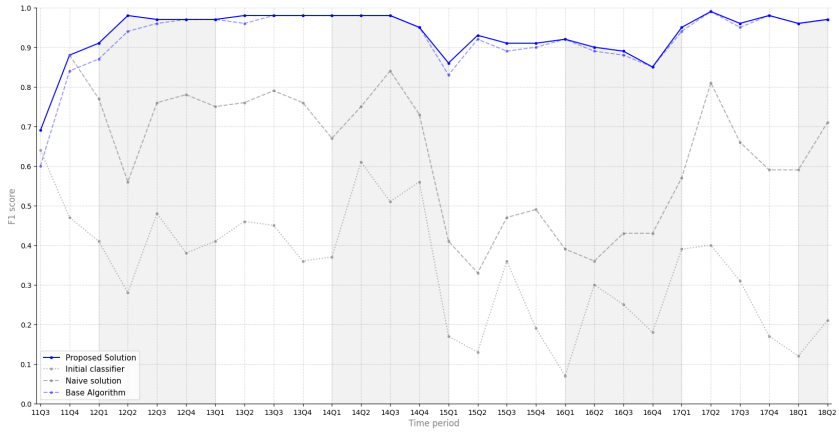


Figure 13: Performance of the proposed solution using the last modification *timestamp [42]*

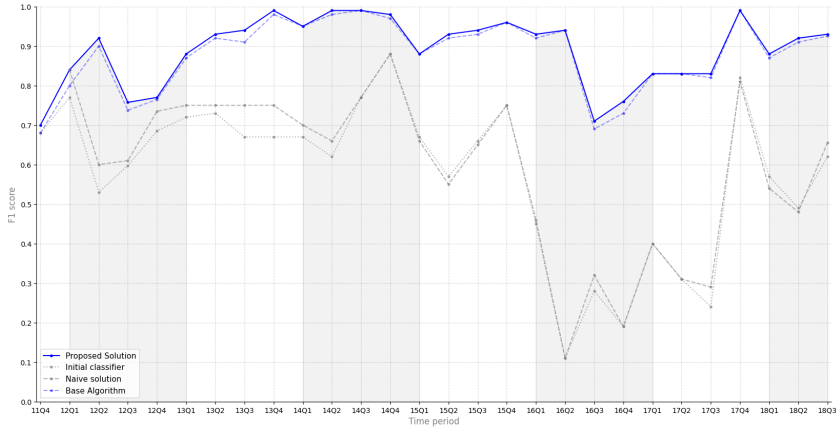


Figure 14: Performance of the proposed solution using the first seen *timestamp [42]*

As can be observed in both figures, the proposed solution outperforms the naive solutions and the original algorithm. The modifications proposed and implemented, even

though no hyper-parameter optimization was performed, enabled the proposed solution to overcome the limitations of the *cold-start* in the initial stages (i.e., up to +10%) and improve the detection quality of the base algorithm in most of the data chunks (i.e., between +1% and +4%). More interestingly, the proposed solution shows great adaptation capabilities and concept drift handling performance over an extended period of time, especially when the *last modification* timestamp is used (i.e., F_1 detection performance over 90% in almost all periods). Although the *first seen* timestamp provided good overall performance, the detection performance is not as smooth and stable as in the *last modification* case. Thus, it is a *less reliable* timestamp to deal with concept drift in Android data. A further exploration of timestamps' properties and their related issues is performed in Publication VI and Publication IX.

The proposed solution, using the *last modification* timestamp, averaged 94.65% F_1 score, 91.17% precision, 94.14% recall, and 80.49% specificity performance metrics in the 7-year-long case study, proving the goodness of the solution to adapt and react to Android malware detection concept drift challenges under imbalanced data conditions. Besides, it outperformed the state-of-the-art solutions *MaMaDroid* [80] and *DroidEvolver* [108], as shown in Figure 15. More specifically, when the training period is excluded (i.e., 2011), the proposed solution averaged 94.05% F_1 score in the 2012–2016 time frame, whereas *DroidEvolver* reported an average of 89.56% for the same time window. However, it is worth noticing that the features and data sets used by these approaches are different. Thus, the comparison is provided to contextualize the goodness of the proposed solution in relation to the state-of-the-art solutions as, due to the different data used, the direct comparison among the approaches is hindered.

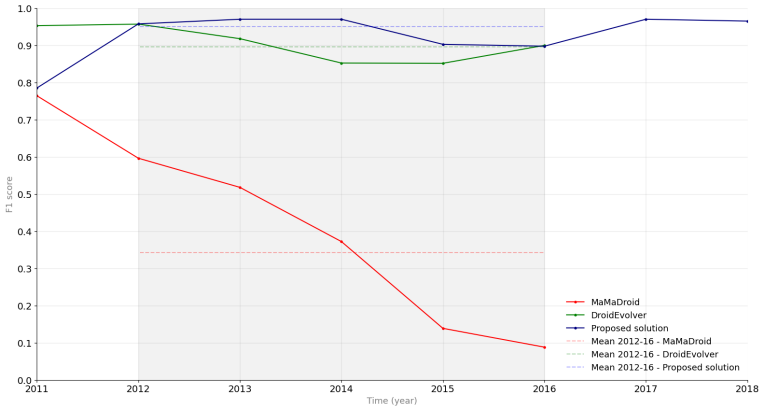


Figure 15: Comparative performance of the proposed solution with state-of-the-art solutions [42]

The proposed solution showed adaptive capabilities to effectively deal with concept drift in Android data using a data stream perspective. Furthermore, when the classifiers dynamics are explored, the proposed methodology can provide relevant insights about the concept drift character and enhance the understanding of the phenomenon. The characterization of concept drift using the proposed solution is explored in the next section.

6.4 Concept drift characterization

The proposed solution can be leveraged to explore thoroughly the phenomenon of concept drift by analyzing the influence of data changes on classification quality measures in various time horizons.

6.4.1 Characterization methodology

For concept drift characterization, the *permutation feature importance* technique [18] was utilized. This method is model-agnostic and applicable in the binary classification case posed by malware detection which can be evaluated by quality measures related to the classification results. The analysis of permutation feature importance scores of chronologically arranged data chunks allows the exploration of changes and observation of the evolution of relevant features in the data, which enable the identification of trends and the characterization of emerging concept drift. The *permutation feature importance* technique is explained as follows.

For a matrix of feature values $\mathbf{X}$ with rows $\mathbf{x}_i$ given each of N observations and corresponding response y_i, $\mathbf{x}_i^{\pi,j}$ is a vector achieved by randomly permuting (π) the j-th column of $\mathbf{X}$. Given a loss function L, the importance VI_j of the j-th feature is defined as the difference between the loss calculated using pseudo-random values and the original data, as it is expressed by the following equation:

$$VI_j^{\pi} = \frac{1}{N}\sum_{i=1}^{N} L\left(y_i, f(\mathbf{x}_i^{\pi,j})\right) - L(y_i, x_i) \tag{1}$$

To analyze the relevant features for concept drift analysis, the permutation feature importance technique was applied to the test data.

The concept drift characterization method used in this study adopts Eq. (1) by the creation of the classification function f_t using data X_t from period P_t. Then, observations X are taken from the set $\biguplus_{l=t+1}^{l+h} X_j$, where h declares an analysis time horizon. For instance, *short term* (i.e., three months), *mid-term* (i.e., six months) and *long-term* (i.e., 12 months) horizons were analyzed in Publication V. The usage of several time horizons enables better characterization of the changes in *importance* of features. The whole procedure is summarized by the following equation:

$$VI_j^{\pi}(t) = \frac{1}{N} \sum_{\substack{i=1, \\ x_i \in \biguplus_{l=t+1}^{l+h} X_l}}^{N} L\left(y_i, f_t(\mathbf{x}_i^{\pi,j})\right) - L(y_i, x_i) \tag{2}$$

The procedure can be used to evaluate the influence of features on various quality functions $Q(.) = 1 - L(.)$ such as F_1 *score*, *specificity* (true negative rate), and *recall* (true positive rate), which are relevant performance metrics on imbalanced data scenarios.

6.4.2 Experimental results

The concept drift characterization technique described above enabled us to analyze relevant issues regarding the evolution of important features (i.e., concept drift), and also understand and compare the system dynamics under each specific timestamp used. A detailed analysis of the latter, which evidenced significantly different classifier dynamics for the timestamps and that the *first seen* timestamp was prone to show temporal delay, affecting the capabilities to deal with emerging concept drift, can be found in Publication V. Publication VI also explores the differences between the timestamps for concept drift handling and characterization, but focusing the analysis on cross-device detection performance. Due to these differences and better suitability of the *last modification* timestamp, the analysis of the evolution of important features was only performed for the *last modification* timestamp. The process is summarized in the following paragraphs.

For each period P_i, the best classifier was selected. The permutation feature importance technique was applied to the classifier using Eq. (2) with F_1 score as loss function.

The *importance* was calculated separately for three test sets (i.e., time horizons). The first set was the subsequent period to P_i, thus P_{i+1}. The second set consisted of the two successive periods, $\biguplus_{j=i+1}^{i+2} P_j$ and the third set contained the four subsequent periods, $\biguplus_{j=i+1}^{i+4} P_j$. As defined, the sets were built incrementally, thus corresponding to three, six, and twelve months data horizons.

The usage of the three incremental test sets enabled us to observe the variation of the importance of features on different time spans. The analysis and related plots, included in Publication V, enabled us to conclude that no feature was found useful or *important* in all periods. A fact that stresses the existence of concept drift in the data. More interestingly, based on these results three types of features were distinguished.

The first type includes those features that are not useful in any time horizon like *getgid32* or *restart_syscall*. Their influence is anecdotic and likely due to the stochasticity of the technique.

The second type of features groups features that are more important in longer time frames (i.e., medium and long term) than in the short-term. These features are not very good at recognizing sporadic threats, but they constitute a solid base in a long-time threat detection system. Features like *clock_gettime* and *flock*, which lie inside this category, show a relatively stable discriminatory power over time.

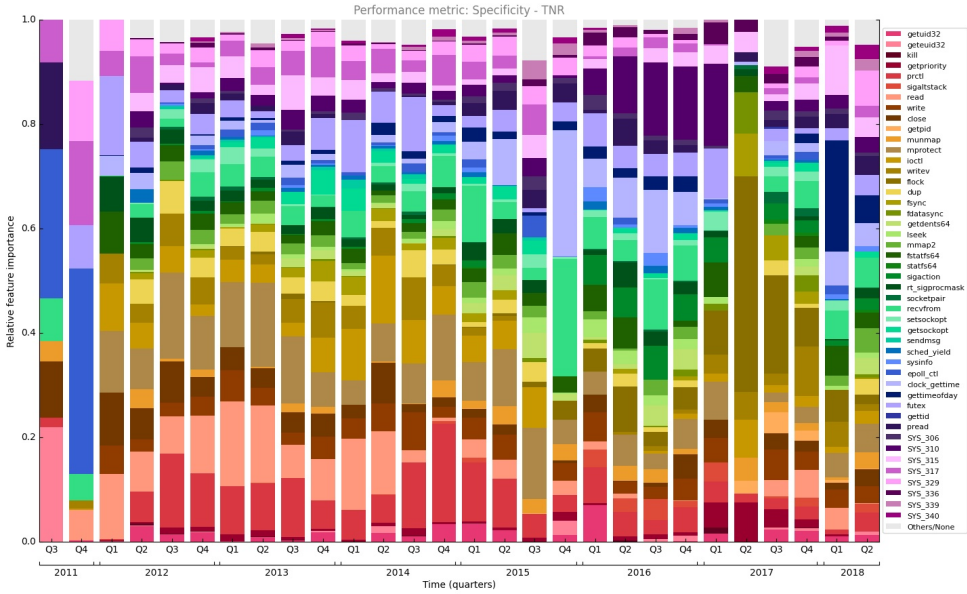
The third type of features presents the opposite situation. The feature is a relatively good discriminator in the short term but is not as useful in longer time frames. These features are less beneficial for overall discrimination than in the short time frame, when a smaller variety of threats is present, because a greater number of unique threats are present in longer time frames (i.e., more families and malware variants). Consequently, these features might work well to distinguish specific malware families. System calls such as *write* or *SYS_317* are included in this category.

To perform a deeper analysis of the importance of features for specific recognition tasks, the permutation feature importance was calculated using *specificity* and *recall* as loss functions. The results for *specificity* provide information about important features to recognize benign software, whereas for *recall*, also called *sensitivity*, they inform about the important features for the malware detection task.

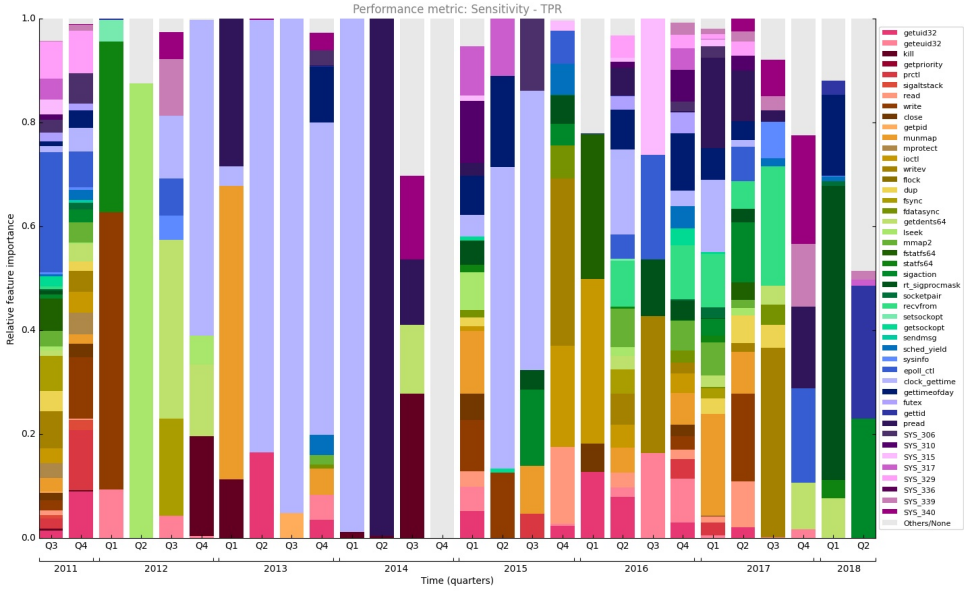
The obtained results are depicted in Figure 16, showing the evolution of important features for the *goodware* and malware recognition tasks, referred to as *specificity* and *recall*, respectively. The horizontal axis provides the timeline, split into quarters or periods. Regarding the vertical axis, the color relates to specific features, while the colored areas (i.e., vertical range) in each bar provide the importance score of each specific feature in relation to the total importance of each specific period of time (i.e., the total importance of a period is the sum of the importance scores of all the important features in that period). Consequently, the larger the vertical range or area spanned by a feature in a bar, the greater the importance of the feature in the specific period.

In the case of the benign software recognition task, presented in Figure 16a, the importance of features appears to be locally stable. Several features like *read* and *mprotect*, depicted with light red and brown colors, respectively, have similar influence for extended periods of time. Besides, quarters with clearly dominant features are rare (e.g., 2011-Q4, 2017-Q2). Despite that some trends can be spotted, with some features gaining importance over time and others losing importance in some periods, the overall picture shows stability and that the same set of features is relevant in all time frames with no distinctive changes in relative influence and with no new clearly dominant features emerging over time.

The results are drastically different for the malware recognition task. Figure 16b shows



(a) Specificity



(b) Recall

Figure 16: Quarterly feature importance scores for recall and specificity [42]

the changes in feature importance calculated for the recall function. As can be noticed, for the majority of quarters, the dependencies observed in a specific period are not repeated in the following periods. Besides, even when a feature shows extremely high importance in one period (e.g., *pread* in 2014-Q2), no consistency is observed and the importance of the feature decreases dramatically in the following periods. The only remarkable exception to this observation is *clock_gettime*, which is a very important discriminatory variable for several years. However, even in this case, there are quarters in this extended time frame where the feature loses completely its discriminatory power for malware detection.

Another issue observed in the malware recognition case is the existence of periods where the total importance of the features included in the bar is far from reaching the top (i.e., 2014-Q4, 2018-Q2). In those periods, none (e.g., 2014-Q4) or few of the included features (e.g., 2018-Q2) were found important for the malware recognition task. In the former case, it suggests that the set of features was not large enough to model all malware types observed in the data, whereas in the latter case, new features emerged as important.

Finally, even though important features seem to vary dramatically among quarters for the malware recognition task, some general patterns can be spotted. For instance, as mentioned before, *clock_gettime* is critically important from 2012-Q2 until 2015-Q2 but not so much after (i.e., more recent years). The internet-related system calls (i.e., *socket-pair recvfrom*, *setsockopt* and *getsockopt*) appear to have more importance for the recent years, from 2015-Q4 to 2017-Q3. More interestingly, the bars from 2012-Q1 to 2016-Q1 show clear dominance of small subsets of features (i.e., mainly *clock_gettime*), whereas in the latter years, the bars are composed of more features, looking more similar to the bars of the benign recognition task.

It is worth noting that, when comparing Figure 16a with Figure 16b, the segmentation of the bars is a major difference between them. For the benign recognition task, the bars are dense, composed of many features, and show stability. On the contrary, the bars for the malware recognition task are mostly composed of a small subset of features, showing clear dominance of some of them over the rest. Consequently, the malware recognition task appears to be significantly more complex and rapidly changing than the benign software recognition task.

The characterization results may aid malware analysts in comprehending the overall evolution of benign and malicious samples and the causes of concept drifts, increasing experts' confidence in learning models. However, despite the goodness shown by system calls to generate an effective detection model, an expert may not derive a clear comprehension of what type of app behavior is induced by each feature as a particular system call can be associated with different system functions. Static features such as permissions or API calls can benefit more from our characterization approach due to a more comprehensible mapping between these features and the application behavior. In this regard, Publication VIII applies the proposed methodology to Android *permissions*.

6.5 Chapter summary

The evolving nature of Android malware has been neglected by the majority of ML-based detection methods proposed in the related literature, thus disregarding the degenerative impact of feature changes over time in the performance of the detection models (i.e., concept drift). This chapter presented a data stream-based approach to detecting,

handling and characterizing Android malware concept drift effectively. The proposed solution adapts to emerging concept drift, enabling long-term effective Android malware detection. Besides, the presented approach allows the characterization of concept driftwhich can be used to comprehend the nature of the changes by security analysts, increase malware-related knowledge and enhance detection.

7 Concept drift and cross-device behavior: implications for effective detection

7.1 The postulate of cross-device consistency

KronoDroid data set provides data collected on two types of devices (i.e., emulator and real device). Both device types have been widely used in behavioral-based Android malware-related research, almost indistinctly. The usage of an Android emulator or a real device usually depends on the access to specific resources and the scale of the study. For example, large-scale studies tend to use emulators as they are easy to deploy, *clean and restart*, and integrate in automated systems. However, they are prone to be bypassed by *anti-sandbox* techniques, suffer from hardware-related compatibility issues and can be fairly limited regarding user interaction and overall phone simulation capabilities (e.g., SIM card). On the other hand, real devices provide full phone operability and user interaction, are immune to *anti-sandbox* techniques, and provide close to *null* hardware-related incompatibilities. However, they are more difficult to deploy, maintain and integrate into automated workflows. Therefore, the selection of the collection device is mainly justified based on purpose, accessibility, or scale matters.

The underlying assumption that enables this freedom of choice is the implicit postulate of *cross-device consistency*. It implies that the behavior of applications is consistent across Android operating systems and device versions, which suggests that the types of devices (i.e., emulators or real devices) and OS versions used do not affect the behavioral profile gathered. Notwithstanding that, this axiomatic cross-device consistency has been challenged by the few studies that worked with data acquired from both kinds of devices.

By design, *KronoDroid* data set is especially suited to assess not only concept drift issues but also cross-device behavioral consistency. This is the focus of Publication VI where the *KronoDroid* data set and the concept drift handling methodology described in Publication V and outlined in Chapter 6 were leveraged to analyze both issues.

7.2 Cross-device behavior and concept drift handling

KronoDroid provides dynamic data collected from the same set of initial applications on two different devices. A simple comparison of the data would be biased and inevitably produce erroneous findings since, as was explained in Section 5.2, the final composition of the two data sets was not the same. For sound experimental comparison, the intersection of the data sets — selected by matching hash value, was employed. Therefore, the final device-related data sets utilized consisted of 34,981 benign apps and 28,343 malware samples.

For the experimental setup, every app within the data sets was described by 288 dynamic features (i.e., system calls count). Even though the available system call set for the emulator is smaller (i.e., 212, named *reduced* feature set), the usage of the whole feature set included in *KronoDroid*, corresponding to the real device which includes 76 additional system calls (i.e., named *extended* feature set), was preferred to perform a sound comparison of the impact of the distinct feature sets sizes on the detection performance. Thus, for the 76 system calls not present in the emulator data, a zero value was imputed for each of these system calls in all emulator apps.

Regarding the timestamps, the *last modification* and *first seen* timestamps were used due to their extensive data coverage and high prevalence among the data samples.

To assess the impact of concept drift in cross-device detection, the methodology detailed in Chapter 6 and summarized in Figure 17 was applied to both feature sets.

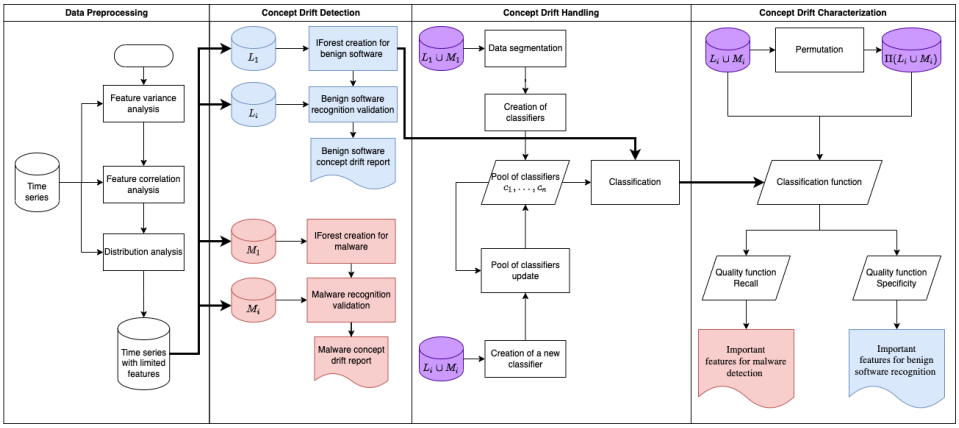


Figure 17: Concept drift detection, handling and characterization scheme [43]

The *data pre-processing* steps applied to each device-related data set resulted in a different subset of features included in the final feature sets for each specific device (i.e., after correlated features were removed). Even though the normality tests confirmed that none of the features was normally distributed for any data set, the distribution plots evidenced distinct data distributions for the same system call on each device. As the samples composing each data set were the same, these differences provided initial support to challenge the assumption of cross-device consistent behavior.

The *concept drift detection* stage evidenced the existence of concept drift on both data sets. More specifically, the selection of the most important features for each initial classifier, reported distinct sets of important features for each data set and timestamp used, as can be seen in Table 11. In this table, the top-10 of most important features for each device and timestamp are provided. The common features in all the approaches are highlighted in blue, whereas the feature sets generated by each specific timestamp are provided with different backgrounds (i.e., grey for the *last modification* and white for the *first seen*).

Table 11: Ranking of the most important features per each data set and timestamp combination [43]

Emulator		**Real Device**	
Last Mod	*First Seen*	*Last Mod*	*First Seen*
rt_sigprocmask	rt_sigprocmask	epoll_ctl	clock_gettime
fcntl64	getuid32	futex	SYS_329
futex	ioctl	SYS_329	writev
getuid32	recvfrom	clock_gettime	epoll_ctl
ioctl	read	writev	getuid32
write	futex	ioctl	write
read	write	write	close
writev	fcntl64	getuid32	gettimeofday
recvfrom	prctl	munmap	ioctl
pread64	fstatat64	read	connect

As can be observed in Table 11, the important features for each timestamp on each specific data set are similar but show different orders. However, when the devices are compared, the feature sets are significantly distinct, with the exception of the 3 common features (i.e., *getuid32*, *ioctl*, and *write*), including architecture-related features in high positions on the list (i.e., *fcntl64* for the emulator, and *SYS_329* for the real device). This

suggested that the timestamp selected might cause differences in the relevant feature set but, more importantly, that the data source can have a critical impact on the definition of the important feature sets.

These initial feature sets were used to induce and test the one-class anomaly detection models in the concept drift detection phase, as depicted in Figure 17. The results for the emulator data set for both timestamps, provided in Figure 18 evidence the existence of concept drift. Even though the results are similar to the ones for the real device data (see Figure 11 in Section 6.2.4), the performance dips for the real device data are deeper than for the emulator data. However, the emulator malware data show more *dips*. In both cases, the *last modification* timestamp generates fewer and shallower dips than the *first seen* timestamp.

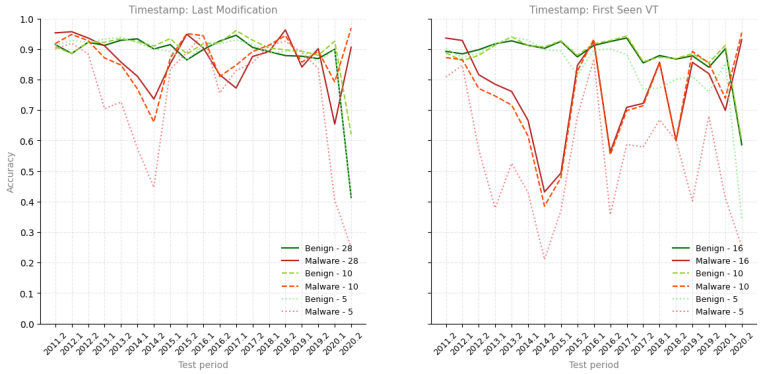
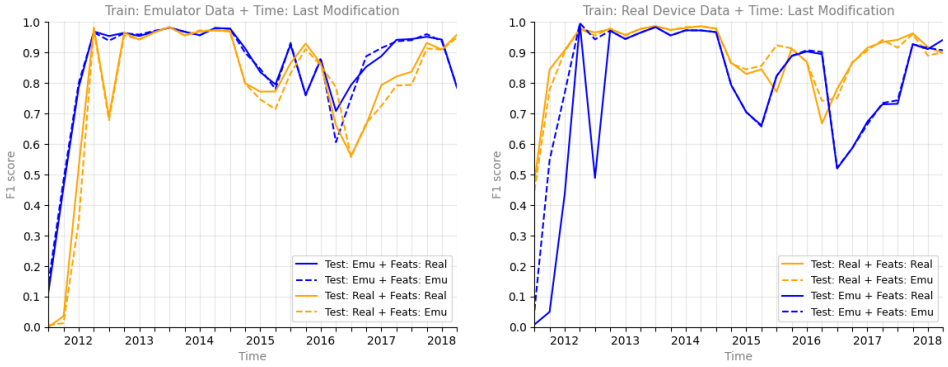


Figure 18: One-class anomaly detection models performance - emulator data [43]

The existence of concept drift on both data sources manifests the need for a handling solution. The solution explicated in Section 6, used to handle concept drift effectively, was used to explore the phenomenon of cross-device detection performance under the presence of concept drift.

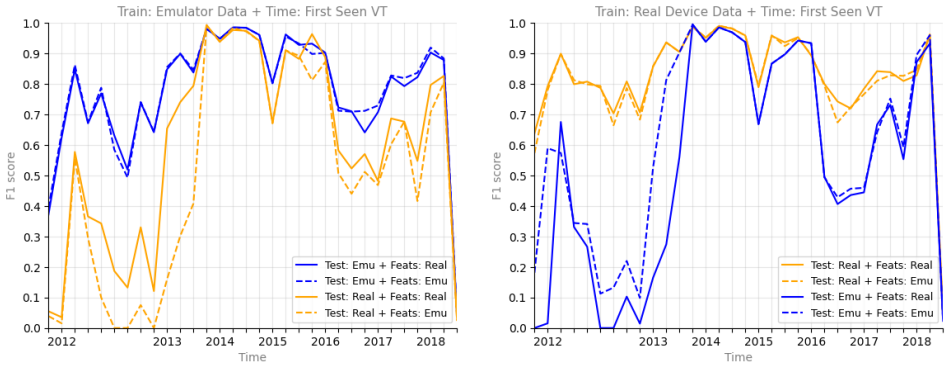
For this purpose, the concept drift handling solution was applied using all possible combinations of training and testing sets described using both feature sets (i.e., reduced and extended feature sets). This enabled us to test the same device (e.g., training and testing with emulator data) and cross-device detection performance (e.g., training with emulator and testing with real device data) using distinct feature set sizes. Besides, the models were induced using both timestamps. This multi-testing scenario allowed us to analyze the cross-device detection performance under concept drift constraints from all possible perspectives. For example, when the emulator data was used as training set using the last modification timestamp, four different combinations were tested by using the two possible feature sets (i.e., reduced and extended) as data descriptors and the two data sources as testing sets (i.e., emulator data for same device detection, and real device data for cross-device detection). The performance results for this scenario on the four distinct testing cases are provided in Figure 19a. Figure 19b provides the performance of the testing scenarios when the training set is from the real device data and the temporal ordering is provided by the last modification timestamp. Figure 20 provides analogous scenarios when the first seen timestamp is used instead of the last modification timestamp to locate the data samples along the Android historical timeline. More precisely, Figure 20a provides the scenario of training data belonging to the emulator, while Figure 20b provides the results for the real device data as training set. As can be observed in all

graphs of Figure 19 and Figure 20, four possible combinations of data are tested, which, depending on the training data used, conveys the performance on the same device data using different feature sets or cross-device performance for both feature sets (i.e., reduced and extended). For the sake of consistency, disregarding the source of the training data (i.e., specified in the title and caption), in all graphs, the color of the lines relates to a specific device data (i.e., blue for emulator, yellow for real device), and the line style informs about the feature set used (i.e., solid for the *extended/real device*-related feature set, and dashed for the *reduced/emulator*-related feature set).



(a) Training with emulator data *(b) Training with real device data*

Figure 19: Last modification timestamp-based detection models performance [43]



(a) Training with emulator data *(b) Training with real device data*

Figure 20: First seen timestamp-based detection models performance [43]

As mentioned before, in the cases where the testing data source differs from the training data source, it enables us to test cross-device performance. The usage of different feature sets provides information about the discriminatory capabilities of a larger feature set versus a reduced feature set.

As can be observed in Figure 19, the performance of the proposed solution is relatively stable (i.e., over 0.80 F_1 score in the plotted time frame) when the *last modification* timestamp is used, and especially when the testing data is from the same data source as the training set. Besides, with just a few exceptions, the results obtained using different

feature sets as data descriptors are very similar. The worst results are obtained for cross-device detection. As a result, this timestamp provides a relatively stable performance, using either feature set, especially for same device data detection. Therefore, the data source has a significant impact on the detection performance of the model, showing that cross-device detection provides poorer and less reliable performance. However, the feature set does not show a significant impact on performance.

When the *first seen* timestamp is used to locate the same set of apps along the historical timeline, the performance and outcomes of the proposed solution change dramatically, as can be observed in Figure 20. This timestamp provided a different data distribution across the timeline, concentrating a significantly larger number of samples in the 2012–2013 period. The cross-device detection performance is significantly inferior to the same-device detection performance, sometimes even approaching null values, with the exception of a specific time frame (i.e., mid-2013–2016). Regarding the feature set, the results show that it impacts the detection performance and that better results are obtained when the *natural* feature set is used, that is, when the feature set is the one related to the training *device*. The relative smoothness of the performance when the *last modification* timestamp is utilized is not observed on the *first seen* performance graphs, which are characterized by abrupt changes from quarter to quarter. This fact may indicate the presence of *artificial* concept drift that, contrary to the *gradual* data drift, expected from the natural changes in the threat landscape, can be hardly modeled and, consequently, cannot be handled effectively using previous knowledge.

The observation of the *artificial drift* caused by the *first seen* timestamp could have been caused by the generation of *historically incoherent* data, that is, the misplacement of data samples across the historical timeline. Historical incoherence occurs when data *originally* belonging to different time frames are blended together and, consequently, generate a not naturally occurring data set. This is opposed to the overall smooth performance observed when the *last modification* timestamp is used, indicating the emergence of a more *naturally* occurring drift in the data. Based on these observations, the analysis of the differences between the timestamps to locate the data samples was well-motivated for further inspection. This divergence was further explored in Publication IX.

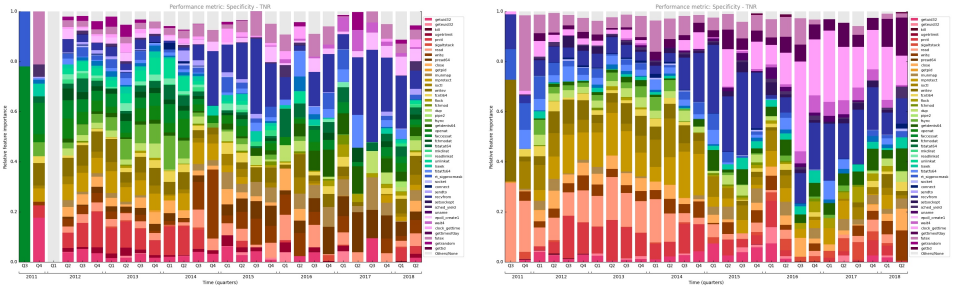
7.3 Characterization of behavioral concept drift across devices

As in Chapter 6, were the concept drift handling methodology was introduced, the proposed solution to handle concept drift effectively was leveraged to characterize the single-device learning models, that is, to describe the important features of the models that use the same testing and training data per analyzed time period (i.e., quarter) using the *last modification* timestamp as sample *context*. The *last modification* was preferred due to the more *natural* emergence of concept drift observed. Permutation feature importance was calculated using specificity and recall as quality functions for both data sources (i.e., emulator, and real device). Figure 21a and Figure 21b depict the important features for the specificity task (i.e., benign software recognition task) for the emulator and real device data, respectively, whereas Figure 22a and Figure 22b convey the same information for the recall task for the emulator and real device data, respectively. For comparison, the same set of features is depicted in the four graphs with the same colors.

As can be observed in Figure 21 for the specificity task, even though the bars for both devices (i.e., quarterly important features) may look relatively similar, as they are densely populated by a wide variety of features in both cases, the comparative analysis shows that a different set of features is important for each device. For example, in the real device, *clock_gettime* (i.e., pink colored) has a greater relative importance than in the emulator

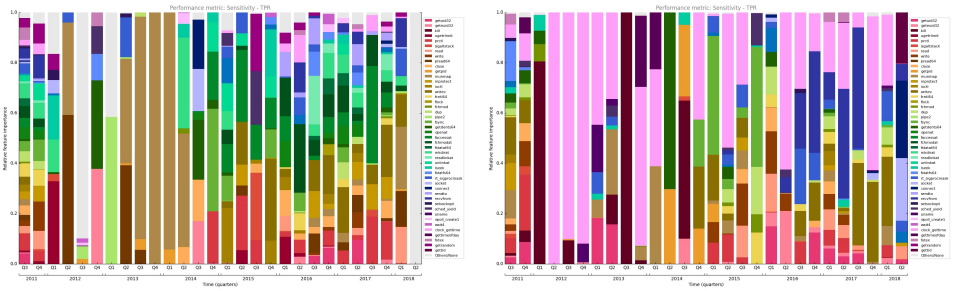
for the last periods. The green areas (e.g., features such as *openat*, *readlinkat*, etc.) are large in the emulator. In contrast, they are negligible in the real device data. For the real device, the brownish features are more important (e.g., *mprotect*), especially in the initial time frames. This fact evidences that, even though the benign data samples use a wide and similar set of features on both devices, the most important features are significantly different across devices.

Regarding the recall task, depicted in Figure 22, a completely different situation is observed. The bars on both devices show a low density of features (i.e., a few features per bar) with clearly dominant features, which are significantly different across devices. For example, *clock_gettime* has a large relative importance in most quarters for the real device, whereas its importance is negligible for the emulator data. In the emulator, *mprotect* and the greenish features provide the majority of the importance in almost all the quarters. For the real device, these greenish features are not significant. As a result, the key features for the malware recognition task change notably between devices across the examined time frame.



(a) Emulator *(b) Real Device*

Figure 21: Important features for the specificity task [43]



(a) Emulator *(b) Real Device*

Figure 22: Important features for the recall task [43]

Finally, when the graphs are compared for the same device (e.g., Figure 21a with Figure 22a), it can be observed that even though the composition of the bars is relatively different (i.e., different distribution of features), the set of most important features is remarkably similar for both tasks, with the predominance of a similar set of features (i.e., similar color tones), but with an extreme polarization towards a smaller set of features for the recall task, in both cases.

The observed differences evidence different behavioral profiles of apps on different collection devices and along the historical timeline, which strongly suggests that the axiomatic *cross-device consistency* cannot be assumed. Further exploration of the issue with statistical significance analysis of the important features between the devices was performed. This statistical analysis is detailed in Publication VI.

The device-based differences in behavioral profiles described in Publication VI, set the ground for a deeper and more fine-grained inspection of the *cross-device consistency* issue. This exploration was performed in Publication VII using a smaller data set but a wider range of collection devices and Android OS versions.

7.4 Chapter summary

This chapter combined and analyzed the implications of two of the main challenges related to Android malware detection: concept drift and cross-device detection issues. The results of the statistical analysis show that data collected on different Android platforms (i.e., real devices and emulators) cannot be detected effectively using cross-device models, as the behavioral profiles for the same set of apps are significantly different. Furthermore, the emergence of concept drift, even when an effective solution to address it is used, magnifies the challenge and greatly impacts the performance of the detection system over time.

8 Cross-device behavioral consistency: benchmarking and implications for effective detection

The behavioral differences observed in Publication VI were further explored in a benchmarking study, described in Publication VII. The objective of this benchmarking was to assess the validity of the cross-device behavioral consistency in a larger set of Android devices. Due to the significant number of variables affecting the behavior of apps at execution time in Android devices (e.g., user interaction, Android kernel, software version) the elucidation of the exact etiology of such differences was out of the scope of the study. The main purpose of this study was to analyze and compare the behavioral profiles acquired for the same set of applications in a large and representative set of Android devices, including different operating system versions.

The generation of effective ML-based detection models requires the usage of a large amount of data. However, for a sound assessment of the validity of the cross-device behavioral consistency, the focus must be placed on the usage of a wide set of collection devices and Android OS versions rather than on the data set size. Besides, the increase in the number of devices and OS versions also increases the likelihood of incompatibility issues which poses an additional challenge for the data set size. The wider the variety of devices (i.e., architectures) and OS versions (i.e., Android API levels), the greater the challenge of finding applications that can successfully be installed and executed on all the devices (i.e., cross-device compatibility), which is a foundational requirement for the soundness and representativeness of the benchmarking setup. As a result, priming data quality over quantity, the data set used in this research was composed of 16 Android apps (i.e., 8 malware and 8 benign samples). The final set of samples was formed iteratively, after the successful installation of samples collected from well-known Android data sets and repositories on all the collection devices. A detailed description of the data samples used in this benchmarking is provided in Publication VII.

A complete *testbed* of Android devices has to include real and virtual collection platforms, as they are both widely used for research purposes. In addition, to analyze the possible differences in system calls on different Android platforms, distinct versions of the OS should be tested and controlled as a possible *confounding* variable. Therefore, for the experimental setup, three real mobile handsets running two different Android OS versions (i.e., Android 9 and 10) were selected as benchmarking devices. The same phone models were also *virtualized* as accurately as possible using *Android Studio's Android Virtual Device (AVD) Manager* and *GenyMotion Desktop* emulators. The *virtual* devices were configured to resemble the real devices' properties and settings as close as possible using the available options in the corresponding emulator software. The real devices were identified with sequential numbering after the R prefix (e.g., *R3*). Their corresponding emulated counterparts were identified with the same number but using a different prefix according to the emulator software used (e.g., *A3* for the *Android Studio* instance emulating the real device 3, and *G3* for the *GenyMotion* instance emulating the real device 3). A detailed description of the devices used is provided in Publication VII.

The workflow of the experimental setup is provided in Figure 23. As depicted, each of the samples composing the data set was installed and executed in all the test devices using two modes of execution. The first mode of execution installed the application, executed it, and let it run freely for five minutes, with no other interaction (i.e., named as 1-event execution or *1E*). The second mode of execution involved the same initial steps but added simulated user interaction in the execution phase. More specifically, 50 pseudo-random events were injected during the run-time (i.e., named as 50-event execution or *50E*. The behavior of the application (i.e., system call traces) was monitored and logged.

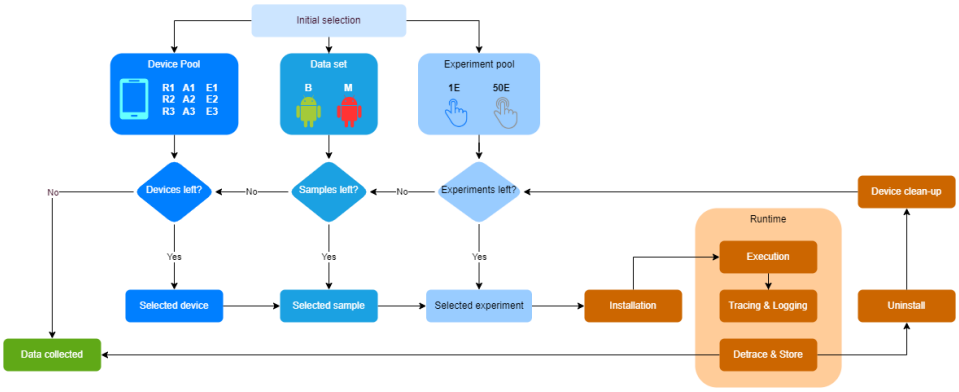


Figure 23: Benchmarking workflow [49]

Therefore, as 16 samples were used in the experimental setup and run on two modes of execution on nine devices, the result of the workflow depicted in Figure 23 yielded 288 behavioral traces. These syscalls traces were further analyzed, being compared for similarity using statistical measures, and used as input for ML models to evaluate the impact of the observed divergences in cross-device and same-device class recognition tasks at small scale.

8.1 Cross-device behavioral comparison

The initial exploration of the raw data logs evidenced differences, with varying proportions, in the length of collected sequences and the number of different system calls invoked by each app during the run-time, for all logs. Even though this observation is expected for distinct data samples, as distinct apps show different behavior (i.e., invoke different sequences and number of syscalls), no significant deviation should be expected for different executions of the same app in distinct devices for the same execution mode if cross-device consistency exists. However, the latter was not confirmed and substantial differences were found in the data logs regarding the behavior of the same app on different devices, for all apps on all devices.

As the data set was composed of 16 distinct apps, 16 distinct behavioral profiles regarding system calls usage were expected since every app differs from the others in its function and nature. To examine the similarity of behaviors of individual apps across devices, the collected data were analyzed for *consistency* and *similarity* across devices. In this regard, feature engineering was used to extract data from the logs and generate meaningful data attributes for comparison. More specifically, the *total number of system calls* invoked by the application and the *number of unique system calls used* were retrieved. Based on these two data features, a more fine-grained comparison of the behaviors was performed using two statistical measures as scoring metrics for each data attribute.

For the *total syscalls* attribute, the *ratio* of increase between the number of system calls produced by the same app on two different devices was calculated. All the calculations were performed pairwise, where the smallest value was always subtracted from the largest. Therefore, the minimum value was 1, implying that an equal number of *total* syscalls was invoked on both devices (i.e., no difference).

For the *number of unique syscalls* data, the system call name was used instead of a *summary* numeric value (e.g., *clock_gettime*). All the comparisons were performed pair-

wise, where the overlap between the unique syscalls sets (i.e., invoked on both executions) was used to calculate the *Jaccard coefficient*. The Jaccard coefficient is a measure of similarity between sets calculated as the size of the intersection (i.e., overlap) over the size of the union of the sets, as expressed in equation 3. It ranges from 0 to 1, and the larger the value, the more similarity between the sets.

$$J(S_1, S_2) = \frac{|\, S_1 \cap S_2 \,|}{|\, S_1 \cup S_2 \,|} = \frac{|\, S_1 \cap S_2 \,|}{|\, S_1 \,| + |\, S_2 \,| - |\, S_1 \cap S_2 \,|} \tag{3}$$

The calculation of these scores resulted in two similarity indexes/scores per sample for each pair of compared devices. For the sake of interpretation, similarity thresholds were established to qualify behaviors as *similar* between devices. The similarity threshold was set to 0.75 for both comparative scores. So that the behavior of a specific app on two different platforms was qualified as *similar* if the total syscalls ratio did not exceed 1.25 and the Jaccard coefficient did not fall below 0.75.

To establish a comprehensible scope, the real devices were employed as the basis for 4 distinct device comparison subgroups. Such division provided distinctive sets of data for comparison. The generated subgroups were coded as *A*, *B*, *C*, and *D*. *Subgroup A* concentrates exclusively on behavioral differences among real devices, while subgroups *B*, *C*, and *D* assess the differences between each real device and its corresponding *virtualizations*. Furthermore, since data acquisition was performed on each device using two classes of apps and two modes of execution, it implied that there were 4 different perspectives to examine the potential contrasts within each subgroup.

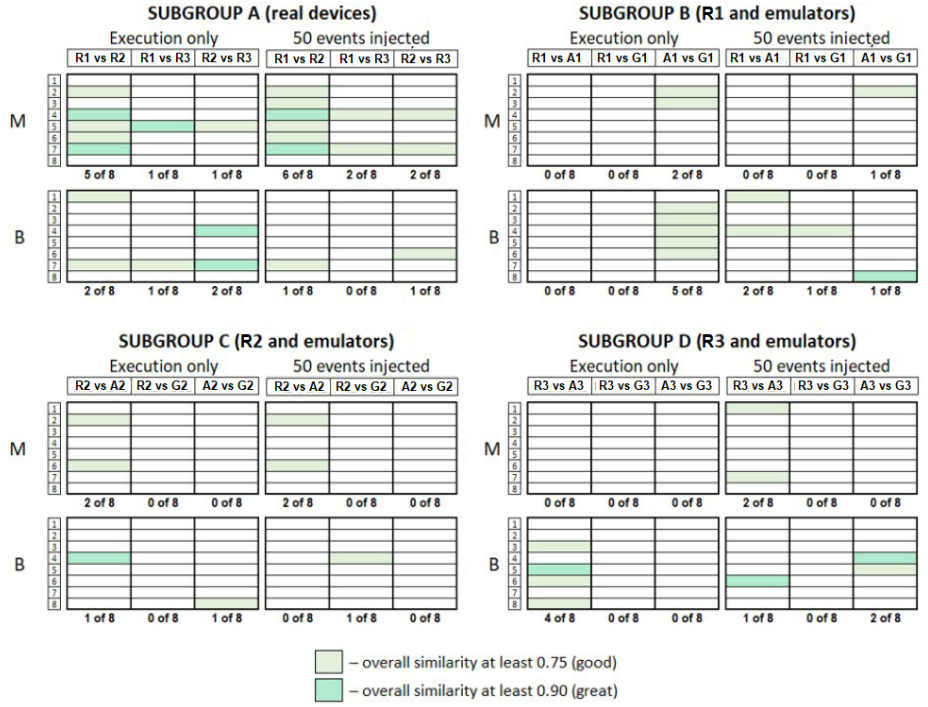


Figure 24: General overview of the comparative results [49]

The general overview of the results, covering the full spectrum of comparison sets and execution modes, is reported in Figure 24. For the sake of the interpretation of the results,

the numeric scores have been omitted, thus providing better visualization and comparison of the observed similarities. The light green colored cells report those devices where the similarity score for the specific app was above the similarity threshold (i.e., *good* similarity). The apps/devices where the sets displayed an outstanding similarity (i.e., the ratio of total syscalls below 1.10 and a Jaccard coefficient greater than 0.90) are distinguished with darker green color.

The results shown in Figure 24 support that, in general, it can be concluded that the behavior of apps is significantly dissimilar across distinct real devices and OS versions. Even though the real devices show similarities for some apps and platforms (i.e., *R1* and *R2*, which belong to the same Android OEM), there is no consistency. The comparatives among real devices and their emulated versions (i.e., subgroups B, C, and D) show remarkably inconsistent results. This fact evidences that the behavioral profiles of the apps in real devices and their emulated versions, even when the virtual devices fully mimic the settings and properties of the real devices, are significantly different.

In general, although some exceptionally similar behaviors were spotted, inconsistent similarity patterns were observed for the vast majority of the analyzed sets under any of the execution modes and both app classes.

8.2 Impact on ML-based detection models

The data acquired in the collection stage were processed to build and evaluate distinct ML-based classification models. More specifically, feature engineering was performed on the acquired data and the absolute frequency (i.e., count) of each individual system call issued by the apps during the collection time was used to describe each application.

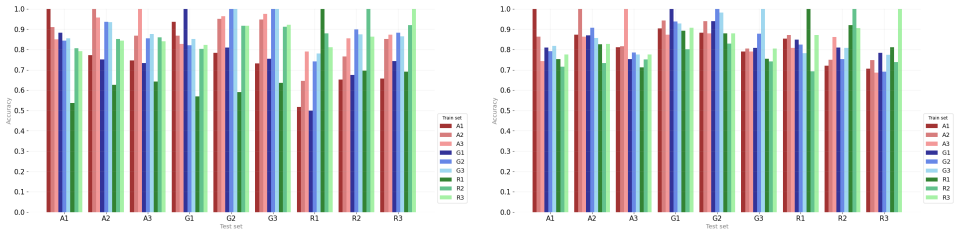
As machine learning models are sensitive to data quantity, the main aim of this experimentation was not the induction of effective forecasting models but the usage of machine learning models to assess the similarity between distinct acquisitions of the same application on different platforms and evaluate the implications of collecting distinct behavioral profiles for the same application in simple detection models. For the sake of consistency, the same classification algorithm and hyper-parameters were used to induce all the ML-based detection models. The models' performances were evaluated using the *accuracy* performance metric.

The underlying concept behind the usage of these detection models was to leverage the model's *overfitting* capabilities to evaluate the similarities between the training and testing set, both composed of 16 instances. In general, an ML classifier model is said to *overfit* the training data when it is trained with limited data, and the resulting model fits too closely or exactly the training data, thus, it does not generalize accurately to *unknown* or *new* data [53]. This is an undesirable situation when building ML models that is usually addressed by using more data or *regularization* techniques. However, in our case, it is leveraged to provide a notion of similarity between the training and testing sets, which are composed of exactly the same samples and described with the same features but collected on distinct platforms. In general, a high-performance ML model should recognize the training data almost perfectly when used as testing data (i.e., training set accuracy). In our case, as the data set is small, it should be perfectly recognizable (i.e., 100% accuracy). Therefore, if an accuracy distinct of 100% is reported, then behavioral inconsistencies can be implied as all the training/testing sets are composed of exactly the same data samples, and the only difference is the collection device, which defines the behavioral profile collected. More precisely, the lower the testing accuracy, the more dissimilar or inconsistent the behavior of the apps on the training device concerning the testing device.

The bar charts displayed in Figure 25a and Figure 25b provide the results for cross-

device detection accuracy for different training and testing sets. The vertical axis provides the accuracy score, whereas the horizontal axis informs about the source of the testing data (i.e., device). Thus, each bar reports the *testing set accuracy* for each trained detection model. The color of the bars informs about the training data used to build the detection model, as specified in the legend. For each execution mode, 9 detection models were induced, trained with each device data, and referenced with the distinct color of the bars. All the trained models were tested separately with the data from all 9 devices, including the training set, and the accuracy performance was retrieved. It is worth remembering that the training and testing sets were composed of the same instances in all cases, the only difference was in the feature values describing each sample which corresponded to the behavior collected on each particular device. The only exception to this fact occurred when the training and testing data belonged to the same device. In such a case, the training and the testing data feature values were identical.

Given the reduced size of the data set and the inherent randomness of the classifier algorithm used (i.e., Random Forest), to achieve a representative measure of the performance, each model training/testing was repeated 100 times. Therefore, Figure 25a and Figure 25b report the average accuracy performance of all models.



(a) Execution-only (1E) data

(b) 50 pseudo-random events (50E) data

Figure 25: Accuracy of cross-detection models [49]

As mentioned before, in our case, the greater the accuracy, the more similar the behavior of the apps across devices and, consequently, the better the discriminatory capability of the model to cross-device data. The lower the accuracy, the more dissimilar the behavior of the apps across devices and the worse class-based discrimination by the model. Note that the discrimination of the training data (i.e., when used for testing) by the model should be close to perfect accuracy to make such implications, that is, the classification model must be able to classify the training data effectively.

As can be seen in Figure 25a and Figure 25b, the accuracy never reaches the maximum score, except when the testing data are the same as the training data. This confirms the goodness of the induced models to perfectly discriminate their own data, but not any other test data, which corresponds to the same data set but is described by the behavior collected on other devices. This demonstrates that even if the samples on the training and testing sets are the same, the data collecting device has an effect on the performance of the induced systems since even in this straightforward scenario, class-based discrimination degrades significantly. With the exception of G2 and G3 in execution-only mode, all cross-device models decrease their performance significantly, showing that the behavior of apps in distinct devices is not consistent, thus confusing the classifiers induced with one device data to generalize effectively to data collected in other Android devices.

In summary, when all models are considered, the cross-device average accuracy for execution-only data is 0.80 (out of 1) with a standard deviation of 0.11, whereas for 50

events is 0.81 with a standard deviation of 0.07. Thus, even though when more events are injected the overall performance does not change significantly, the behavior appears to be slightly more consistent across devices, showing lower variability. Therefore, the behavioral differences captured on the testing sets with regard to the training set impact the model's performance significantly.

In conclusion, the classifiers' cross-device detection performance is significantly inferior to same-device data detection performance because of behavioral variations among devices.

To further explore the implications of mixing behavioral profiles from distinct collection platforms, *mixed* models were induced using training sets constructed using random combinations of data from distinct devices. In this scenario, the benign and malware data were independently and randomly selected from one of the nine data sources, which ensured that the data set was always composed of the 16 different samples. The trained model was tested, as in the previous case, with data collected from all devices. Therefore, the training sets included class samples from randomly selected devices, whereas the testing data always belonged to a single device.

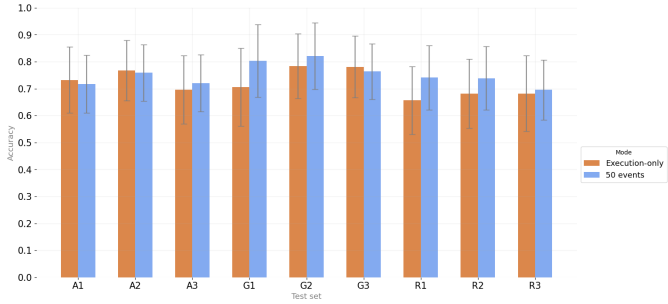


Figure 26: Mixed *models performance results [49]*

For this experiment, a total of 10,000 models were induced. The average accuracy results are provided in Figure 26 for both execution modes (i.e., orange for execution-only and blue for 50 events). The height of the bars reports the average accuracy value, while the standard deviation is depicted as extended grey lines below and above the average value. As the horizontal axis reports the source of the testing data set, the accuracy score informs about the discriminatory properties of each data source for mixed data models.

As shown in Figure 26, the performance of the models where the training data is mixed is remarkably lower than for the models induced in Figure 25a and Figure 25b, where the data for both classes were collected from the same device. More specifically, for the execution-only mode, an average accuracy of 0.72 is achieved with a standard deviation of 0.044 whereas in the 50 events mode an average accuracy of 0.75 is reported and a standard deviation of 0.038. More interestingly, the data collected on the *Genymotion* emulator (i.e., *GX*) emerge as the most easily recognizable by the mixed models, whereas the data collected in the real devices are the most challenging and report the worst performance of all mixed models. In this case, 50 events data are easier to discriminate than execution-only data. This may suggest that more events would tend to make the behaviors more similar across devices and that, consequently, are easier to discriminate by the mixed models.

In any case, despite the existence of some limitations, described in Publication VII, these results evidence that cross-device consistent behavior cannot be assumed and that

the data source must be considered in the design and data pipelines of any robust ML-based Android malware detection system. Furthermore, mixing data from distinct sources in both training and testing sets seems to impact notably the performance of the classifiers. Our results show that on average, in simple models and easy data sets, the cross-device accuracy of single-source trained models might be $\approx 20\%$ lower than the same-device testing accuracy and $\approx 30\%$ lower in the mixed models case.

8.3 Chapter summary

Most of the research regarding Android malware detection assumes some form of *behavioral consistency* of applications across Android devices. Consequently, the impact of the nature of the devices and operating system versions used in the collected behavioral profiles has not been considered. In this chapter, the cross-device consistency issue, introduced in Chapter 7, was deeply explored by performing a thorough benchmarking of Android apps in a wide set of Android devices. The results confirm that cross-device behavioral consistency cannot be assumed as the collected behavior of the same set of apps in different Android platforms and OS versions differed significantly. Neglecting this issue may lead to a severe decrease in the performance of the detection models designed for specific platforms when facing data collected on a different platform. Cross-device detection is a challenge that must be considered and cannot be assumed on the basis of behavioral consistency across platforms when system calls are used as detection features.

9 Leveraging the first line of defense against malware: Android security permissions

The previous chapter and related publications dealt with behavioral attributes of Android apps (i.e., system calls). Dynamic analysis of Android applications is a time-consuming and specialized task. It requires setting up a *sandbox* environment, using a sophisticated analysis toolkit, and running the application for a specific amount of time. As compensation, these features are relatively immune to obfuscation and encryption attacks which tend to bypass static approaches. Despite that, most of the Android detection systems proposed in the literature are built on static attributes. In general, these attributes are easy and fast to collect and may enable on-device detection. *Permissions*, *API calls*, and *intent filters* are widely used static input features for Android malware detectors.

The data collected for *KronoDroid* data set, described in Chapter 5.2, includes, among other static attributes, features related to Android *security permissions*. *Permissions* are the most widely used static attributes in Android malware detection research. Besides, permissions are a built-in security feature, based on the Linux kernel, that constitutes the first line of defense against malicious threats on Android devices.

9.1 Permissions evolution and concept drift handling

Publication VIII explores the application of the proposed methodology to handle concept drift in Section 6.3.1 to the feature space defined by the permissions feature set. Contrary to the system calls feature set, which has remained relatively stable over time, the permissions feature set has evolved significantly over time, as permissions have been added or deprecated in almost every new Android API release.

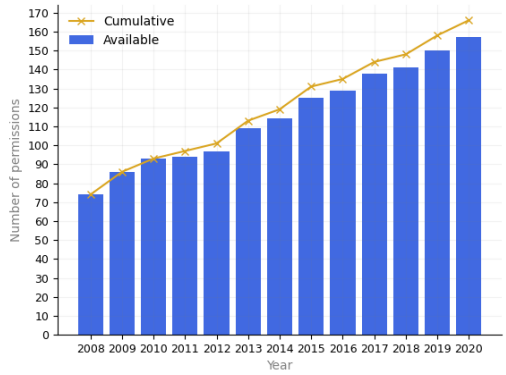


Figure 27: Android permissions timeline evolution [38]

The changing nature of the permissions feature set is depicted in Figure 27. The initial set of *standard* permissions was composed of 74 permissions (i.e., Android 1.0, API level 1, released in 2008). Since then, the permissions set has evolved to contain 157 available permissions in 2020 (i.e., Android 11, API level 30). However, a total of 166 permissions were defined and available at some point, with 9 of them deprecated along the process, thus not supported by the release of Android 11. Not only the addition and deprecation of permissions makes the feature set dynamic, and constantly changing, also the changes in trends and behavior of apps over time cause an impact on their prevalence, and, conse-

quently, in their usefulness as discriminatory features. As a consequence, the detection models based on permissions are prone to concept drift issues, thus requiring to handle effectively the emerging drift to maintain high detection performance over time.

9.2 Experimental results

The concept drift handling method and workflow proposed in Section 6.3.1 was applied analogously for categorical input data (i.e., permissions) as for numeric data features (i.e., system calls). The only differential steps were performed in the data pre-processing stage regarding the data set and the feature selection method used. The methodological nuances regarding the pre-processing stage, and the overall results for concept drift handling and characterization using *permissions* as input features, covered in Publication VIII, are summarized in the following sections.

9.2.1 Data set and feature sets

KronoDroid data set provides a different number of samples per sub-dataset, according to the dynamic data source. As permissions are static features, their values do not differ between collection platforms. For that reason, to use the largest data set possible, both *KronoDroid* sub-datasets (i.e., real device data and emulator) were merged. After the removal of duplicated samples, the resulting data set was composed of 78,804 Android apps, i.e., 37,020 benign samples and 41,784 malware.

Regarding the features used to describe the apps, the permission-related indicators (i.e., binary features) were retrieved. As depicted in Figure 27, until API level 30, 166 permissions were defined. Therefore, each app within the data set was described by 166 categorical features related to permissions attributes, a timestamp, and the class label.

The pre-processing steps applied to the input features, as described in Section 6.3.1, showed that 26 permissions from the initial feature set had null variance and 18 were *strongly* correlated with another feature (i.e., Kendall's $\tau > 0.80$). Therefore, the resulting feature set was composed of 122 permissions. This feature set was named as *full* feature set. In order to test the bias of zero-filled values for non-available permissions at specific times and the impact of permissions evolution, a reduced feature set was formed by applying the pre-processing steps to the set of permissions defined in API level 1. The *reduced* feature set was composed of 60 permissions.

9.2.2 Concept drift handling

The performance results of the proposed solution to handle concept drift as detailed in Section 6.3.1 using *permissions* as input features are provided in Figure 28. For the sake of comparison, recall, specificity and F_1 performance metrics are reported in Figure 28, and detailed in red, green, and blue color, respectively.

As can be observed, the detection system provided over 0.91 F_1 scores in almost all quarters, averaging 0.93 F_1 score in the analyzed time frame using both feature sets. No significant differences were found in the performance metrics using both feature sets, thus proving the goodness of the *reduced* or initial feature set to discriminate effectively Android apps over time. Therefore, in the analyzed time frame, spanning 7 years of Android history, the usage of more features did not enhance the results. The added permissions in later Android OS releases (i.e., API levels) do not seem to significantly impact the performance of the model. In this regard, an in-depth analysis of the evolution of the importance of the features over time is performed in Publication VIII, and summarized in Section 9.2.3.

The specificity metric of the detection system remained relatively stable for the whole

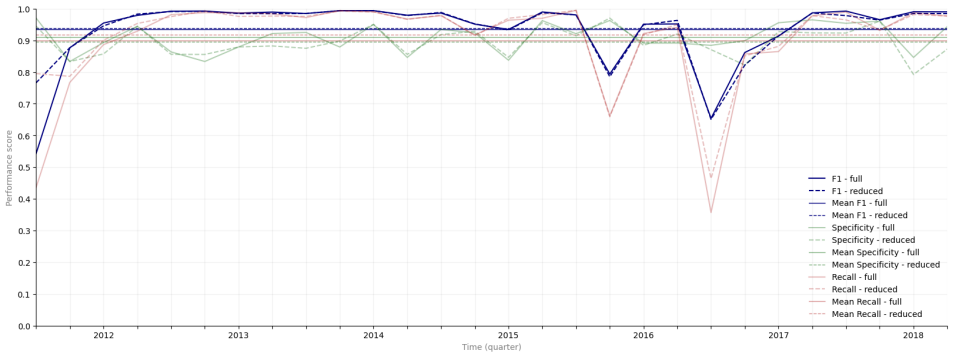


Figure 28: Performance of the proposed solution using permissions as input features [38]

time frame (i.e., over 0.80), whereas the recall performance dipped significantly in three quarters (i.e., 2011-Q3, 2015-Q4, and 2016-Q3). Even though the average of both metrics is high for the whole period (i.e., over 0.90 in both cases), it is worth noticing that the detection performance varies between feature sets for class-based detection. For example, the *reduced* set provides better performance than the *full* feature set to detect malware (i.e., recall), whereas, the *full* vector yields better detection performance than the smaller subset for benign software detection. The characterization of concept drift provided next enabled us to explore the reasoning behind these observations.

9.2.3 Concept drift characterization

The analysis of the evolution of the importance of permissions enabled us to explore the reasons behind the *recall* drops and the class-related performance differences between the feature sets observed in Figure 28. The results and main findings, explicated in Publication VIII, are briefly summarized in the following paragraphs.

The **feature importance evolution** analysis was performed independently for the *full* and *reduced* feature sets. The latter enabled us to analyze the impact of the initial set of permissions per quarter along the whole timeline, whereas, the former, the significance of the later additions in the updated detection models. For the sake of brevity, only the results regarding the *full* feature set are provided for the benign software recognition task and the malware recognition task in Figure 29 and Figure 30, respectively. In these figures, the *relative* feature importance (i.e., the share of the total importance of the chunk provided by each specific feature) is depicted per quarter in vertical bars. The colored areas in each bar are associated with specific permissions. Finally, the white overlay line provides the maximum importance value, provided by the most important feature, in each quarter chunk. For the sake of interpretation, Figure 29 and Figure 30 only depict the permissions included in the *reduced* set with colors. The permissions added in later stages, thus belonging only to the *full* feature set, are masked in grey for each quarter.

Relatively mild and *gradual* concept drift is observed for specificity, with the features varying in importance over time, especially for the last years, as can be observed in Figure 29. More interestingly, the same set of features belonging to the reduced feature set has retained most of the importance over time. The later additions show a limited local impact in specific quarters as depicted by the small grey areas. The dominance of the reduced feature set is further emphasized by the large importance value shown by the most important feature per quarter, always belonging to the reduced feature set, reported by the overlay white line.

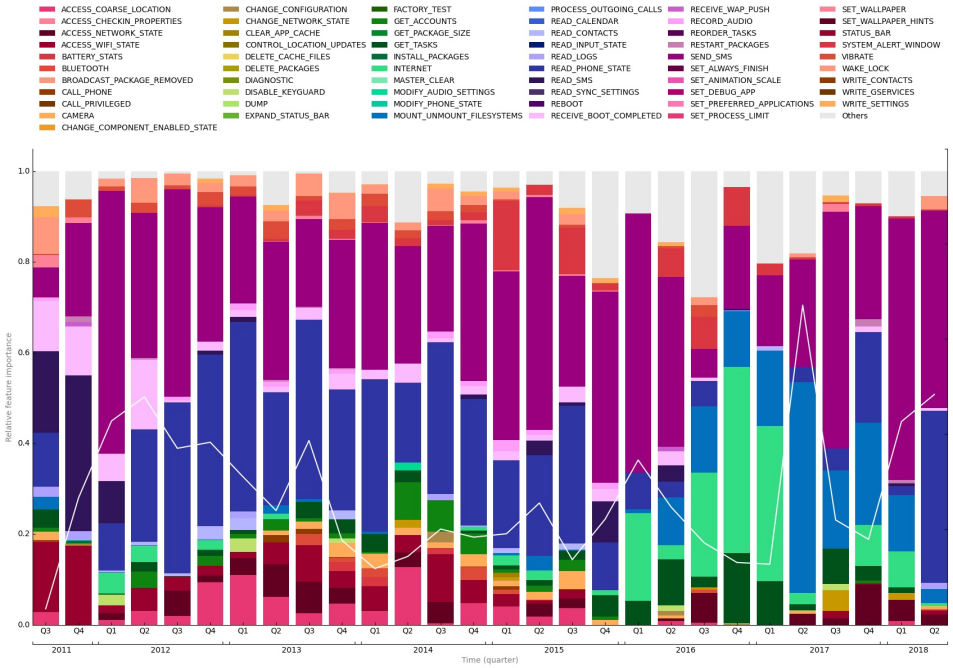


Figure 29: Quarterly feature importance for specificity [38]

For recall, depicted in Figure 30, the situation is dramatically different. This recognition task is characterized by sudden concept drifts, more important features per quarter, and remarkably distinct important features for the last quarters than in the early ones. Besides, the importance of the later added permissions is very significant in some quarters, explaining over 50% of the overall importance and even becoming the most important feature (see Publication VIII). A fact that is not observed for the specificity task. Lastly, the maximum importance per quarter is always small, rarely over-passing the 8%.

Based on the spotted differences between both tasks, the recall task's complexity is deemed more challenging and varied, less stable over time, and more susceptible to sudden concept drift. The observed changes in the last periods demonstrate that malware is more unpredictable than benign software in permissions usage.

In summary, the characterization and analysis performed evidenced the critical importance of the initial set of permissions to build an effective recognition system and the lower relevancy for such a purpose of the later added permissions. Even though concept drift issues were found in benign and malware data, the former shows relative stability with gradual changes, being relatively easy to address, whereas the latter is characterized by more sudden, and complex concept drifts dominated by specific features, making it a challenging task. Besides, the set and degree of importance of features differ for both tasks. Therefore, the analysis performed evidences the dynamism and constantly evolving nature of the malware threat landscape, and emphasizes the critical requirement to address concept drift for any solution aiming to provide long-lasting effective malware detection. The detection solution must have the ability to adapt, updating its knowledge, in an ever-evolving landscape. A constant change that has been overlooked by the permissions-based solutions in the related literature.

A thorough analysis of the **class-based recognition performance**, that is *recall* and

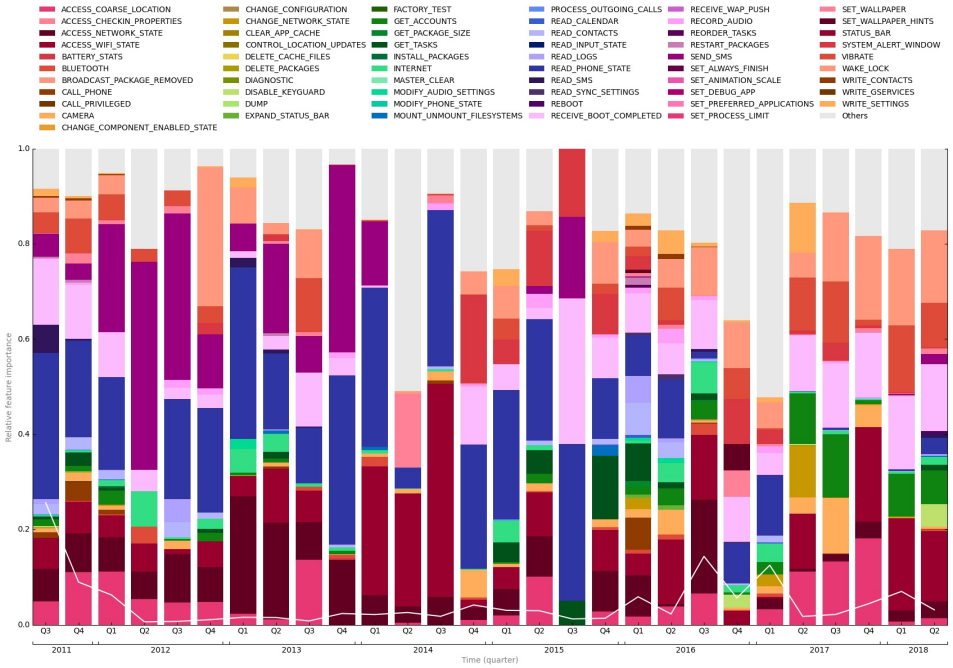


Figure 30: Quarterly feature importance for recall [38]

specificity metrics per quarter, as depicted by the red and green lines in Figure 28, evidenced that the detection system provided high specificity consistently (i.e., over 0.8 in all quarters) using either of the feature sets. This fact emphasizes the goodness of the permissions-based model to recognize benign apps effectively over time and is consistent with the *smooth* concept drift that characterized these data. However, the situation is notably different for *recall*. Even though the average recall performance of the system shows an accuracy over 0.90, even reaching 0.99 in eight periods, the malware recognition performance dips significantly in three specific time frames. The algorithm could not accurately identify new malware samples in those quarters while maintaining high specificity. Figure 28 demonstrates that the reduced feature set is superior to the complete feature set for long-term accurate malware detection since the full feature set's average performance line is lower than the reduced feature set's average line (i.e., a difference of 1.8%). Besides, the third dip was less severe for the model that used the reduced feature set. Therefore, for malware detection purposes the reduced feature set is preferred. The opposite situation happens in the case of benign software recognition, where the full feature set provides better average performance than the reduced feature set (i.e., a difference of 1.5%). This observation is consistent with the results of Publication IV where benign data samples were found to use a smaller but more varied set of permissions than malware apps. Consequently, the extended feature set, as it includes more permissions, provides better overall performance for this task.

Despite the overall high performance of the detection system on both tasks, the system provided diminished malware detection performance, with different degrees of severity, at three specific time frames, namely, 2011-Q3 (initial period), 2015-Q4, and 2016-Q3, as shown by the red line in Figure 28. Publication VIII explores with great detail the etiology behind these dips. The main findings are summarized in the following paragraphs.

The first dip happens in the initial period, 2011-Q3. Even though this should not be considered a dip in performance, it is worth analyzing its cause. In the initialization phase of the system, where the model has access to a limited amount of data, the performance of the system strictly depends on the generalization capabilities of the initial data chunk concerning the following quarters. Therefore this dip does not relate to the learning capabilities of the system as there is no previous reference of detection performance. In our case, the initial chunk was split into n ordered data chunks, where n refers to the number of classifiers in the pool of classifiers. The system was initialized in this initial quarter, building a distinct model with each data chunk and incorporating it into the pool. As a result, the lower initial performance is likely caused by an insufficient variety of data to capture the phenomenon properly. Despite that, the malware detection performance increased in the subsequent quarters, demonstrating the capabilities of the model to learn and adapt over time, even when a distinct combination of features emerged as important in closer quarters (e.g., 2012-Q2).

The second and third dips correspond to actual *sudden* concept drift. The relation between these dips and the characterization graphs (i.e., Figure 29 and Figure 30), is summarized as follows.

The second dip (i.e., 2015-Q4), which is less severe than the third dip, was caused by a sudden malware shift (i.e., new features emerge as important, changing the distribution of the important features significantly, and the previous important features see their influence diminished) coincidental with an increase in the specificity score. Despite these learning challenges, the F_1 score of the system showed an acceptable performance (i.e., over 0.80), improving in the subsequent chunks.

Figure 31: Malware family distribution per period [38]

The third dip (i.e., 2016-Q3) is the major performance drop in the analyzed time frame. For recall, the feature importance distribution shows a remarkably different character

than in the previous quarters (i.e., more features becoming important), with the previously most important feature having a marginal role and the emergence of three new features as important (i.e., ACCESS_NETWORK_STATE and ACCESS_WIFI_STATE from the reduced feature set and READ_EXTERNAL_STORAGE from the extended feature set). This correlated with the maximum value in the absolute importance line. A similar situation is observed for specificity, with old features losing importance and new features emerging as critically important. These sudden and significant unforeseen changes in importance from previous patterns did not enable the pool to deal properly with the new data characteristics. However, after this dramatic decrease in recall performance, where the system still kept acceptable benign software recognition capabilities, the pool adapted and learned from the new data, improving and rapidly recovering past performance levels in the subsequent chunks. Despite that, this dramatic decrease in recall deserved further exploration. This investigation was performed via **malware family evolution analysis**. In this regard, Figure 31 shows the distribution of the top 10 malware families per quarter in the analyzed time frame. The graph shows the proportion and prevalence of 54 malware families over time. It also reports the number of malware families per period (i.e., white stars in the middle of each bar). Besides, the F_1 score of the detection system is provided as a reference in an overlay white dashed line.

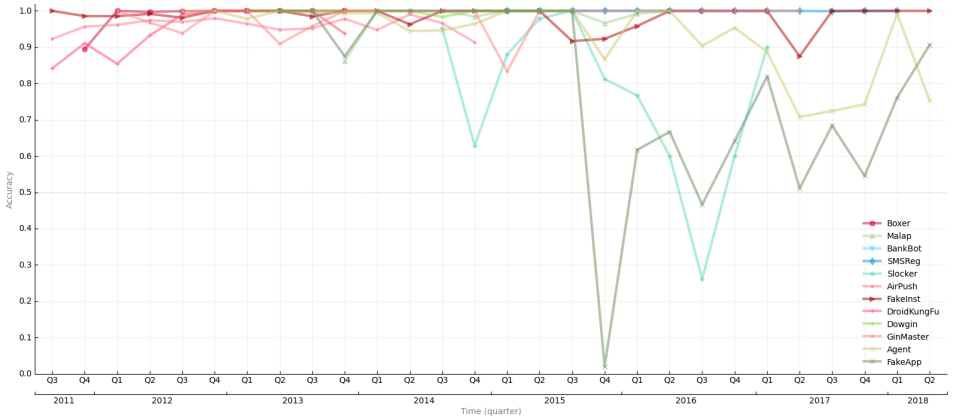
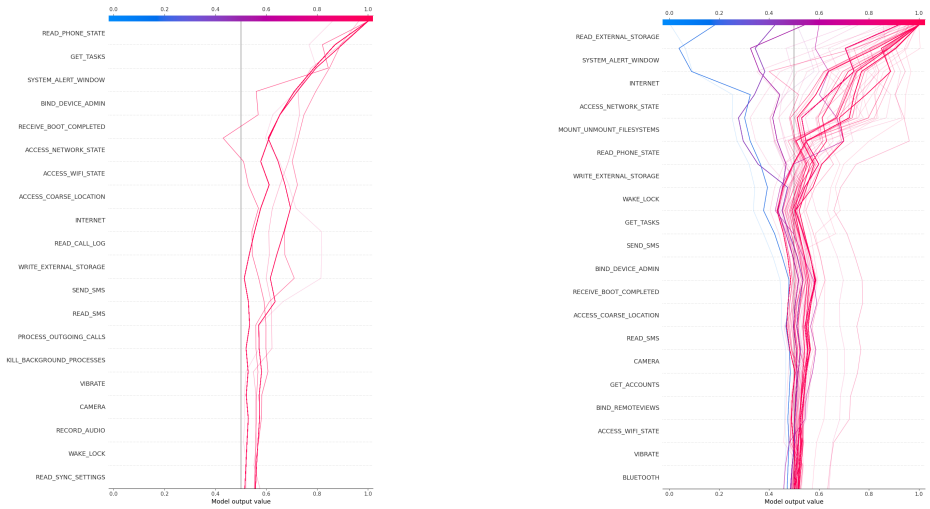


Figure 32: One-family *anomaly detection models performance [38]*

As can be noticed, the major dip occurs in a quarter dominated by the *Slocker* malware family (i.e., the first and most relevant Android ransomware). This suggests that the dip was likely caused by a diminished ransomware detection capabilities of the system. However, this malware family was also dominant in 2015-Q3 and the detection system reported 99% recall. This motivated the exploration of *intra-family* evolution. *One-class* anomaly models were used for such a purpose in which the malware family was leveraged as the *class* concept. These models enabled us to analyze malware family changes over time from an *initial* or *breakout* quarter. More precisely, the decrease in performance of these anomaly detection models indicates a shift in the characteristics of malware family samples (i.e., changes from the initial data used to induce the anomaly model).

Figure 32 provides the *one-family* anomaly detection models for 12 predominant families in the time frame from 2011-Q3 to 2018-Q2. The initial models for each malware family were induced in distinct quarters when the malware family produced an outbreak or their prevalence was significant, and they were tested with data from subsequent quarters until samples belonging to the malware family vanished. This figure provides relevant insights

about malware families' evolution that are directly linked to the performance dips and observed concept drifts. Firstly, most of the malware families showed similar or the same character over time (i.e., high detection accuracy per quarter). This implies that most malware families did not change much regarding permissions over time, thus making these features powerful discriminators. Secondly, the *Slocker* family did not follow that pattern. The initial model built for the *Slocker* family dips significantly in 2016-Q3, suggesting a dramatic and sudden change in characteristics concerning the initial model's samples. This experimental finding was confirmed by reports from threat intelligence sources [87, 106], stating that in the second half of 2016 over thrice *Slocker* variants were detected in comparison with the same period in 2015, and the occurrence of a recursive ransomware outbreak characterized by its evolution into a more sophisticated and diverse malware family. This last fact was also suggested by the more complex and diverse quarter characterization depicted in Figure 30.



(a) 2015-Q3 Slocker *predictions decision paths*

(b) 2016-Q3 Slocker *predictions decision paths*

Figure 33: Slocker *family 2015-Q3 and 2016-Q3 predictions decision paths [38]*

Additional evidence of this diversification of the *Slocker* ransomware family is provided in Figure 33. In this figure, *Shapley values* were used as *local* interpretation method to plot and analyze the individual decision paths leading to prediction outcomes by the model. The decision paths enable the understanding of the importance of features in the decision taken by the model with respect to specific samples (i.e., malware or not malware). More precisely, for each sample, a decision path is generated that starts in an initial feature (i.e., bottom of the graph) and an initial probability of ≈ 0.5 to belong to the malware category and ends in the upper part of the graph with the last feature and providing the malware class probability assigned by the model to each specific sample. The line or *decision path* leading from the bottom to the upper part of the graph, moving along the features placed in the Y-axis ordered by *importance*, depicts graphically the impact of the features on the decision probability along the path (i.e., increasing or decreasing it) until the final probability is reached.

In this regard, Figure 33 compares two relevant quarters dominated by ransomware samples (i.e., 2015-Q3 and 2016-Q3) and the model decisions for the ransomware sam-

ples in each specific quarter. As can be seen in Figure 33a, all ransomware samples were correctly classified by the model and the decision explanations are similar for all samples with just six decision paths explaining all of them. The ransomware samples in this quarter thus exhibit similar traits, and as a result, similar attributes are utilized to classify them. On the contrary, Figure 33b shows numerous decision paths used to classify the ransomware samples, with varying features importance, and many misclassified samples reaching the no malware outcome (i.e., blue paths). Therefore, the *Slocker* samples from 2015-Q3 are significantly distinct from the 2016-Q3 samples, both in variety and characteristics. A deeper analysis can be found in Publication VIII for the *Slocker* family, which explains the performance dip in 2016-Q3, and also for the *FakeApp* family which is related to the lighter dip observed in 2015-Q4, as suggested by the performance of the *one-family* models for this family in Figure 32.

9.3 Chapter summary

This chapter presented the results of the application of the concept drift-handling methodology proposed in Chapter 6 to the permissions feature set. Permissions are *first-line* security constructs in Android OS, inherited from the Linux kernel. The analysis of their evolution and usage can provide relevant insights into the dynamics of the malware threat landscape. Permissions are usually mixed with other static or dynamic features as input features in the detection systems proposed in the literature. Our findings show and characterize the concept drift that exists in the permissions feature space, but more interestingly, they show that when concept drift is addressed, a limited set of permissions that were established in the early days of Android can still be used by themselves for reliable malware detection over time.

10 On the relativity of time: a study of timestamps for effective Android concept drift handling

The essential constructs underlying effective concept drift handling are *timestamps*. Timestamps enable the temporal location of apps, aiming to provide a reliable temporal context within the Android historical timeline. The reliability of many produced timestamps can be questioned, however, because of how malware is created and discovered and the lack of a clear method that can guarantee absolute precision and accuracy.

Publication IX thoroughly explores the usage of distinct *timestamping* approaches for effective Android concept drift handling in different feature spaces (i.e., system calls, permissions, and API calls). The timestamps analyzed are compared using distinct statistical measures and the detection performance yielded using the methodology proposed in Publication V for concept drift handling.

10.1 Data set: timestamps and feature spaces

The base data set used for this study was *KronoDroid*. It provides four timestamps and two commonly used feature spaces (i.e., system calls and permissions). For this benchmarking study, the real device data set was used, composed of 41,382 malware samples and 36,755 benign samples. Additionally, for all the samples in the data set, two timestamps (i.e., *dex date* and *manifest date*) and the API calls defined in the source code (i.e., static features) were collected for the exploration and benchmarking performed in this study. Publication IX provides detailed information on the methodology used for the acquisition and generation of these data features. The timestamps used, their acronym, and brief definition are provided in Table 12. The name of the feature spaces investigated along with their type and dimensionality are reported in Table 13.

Table 12: Summary of the timestamping approaches analyzed [37]

Timestamp name	Acronym	Description
Last modification	LM	The most recent timestamp retrieved from any file inside the apk archive
Earliest modification	EM	The oldest timestamp retrieved from any file inside the apk archive
First seen	FS	Date and time of the first submission of the sample to *VirusTotal*
First seen in the wild	FSW	Date and time of the first time the sample was seen anywhere on internet
Dex date	DD	Timestamp retrieved from the *classes.dex* file (i.e., apk compilation time)
Manifest date	MD	Timestamp retrieved from the *AndroidManifest.xml* file inside the apk

Table 13: Summary of the feature spaces explored [37]

Feature space	Dimensions	Type
System calls	288	Numeric
Permissions	166	Binary
API calls	53,523	Binary

The feature spaces analyzed in this research for the same data set are representative of the most common attributes used for Android malware detection [90] and provide complementary perspectives for the concept drift phenomenon in varying dimensions and feature types. In addition to that, the temporal dimensions segment and transform the multi-dimensional feature spaces distinctively, thus providing an extensive exploration of the suitability of the timestamps for effective concept drift handling.

10.2 Timestamps statistical analysis

The main focus of this study is on the temporal dimension of the data and its impact on concept drift representation, analysis, and handling. Therefore, the initial step was to perform a comparative analysis of the six timestamps using statistical metrics.

The usage of a timestamp to locate applications along the Android historical timeline is subject to *availability* and *reliability* issues. The first pertains to the timestamp's accessibility, while the second concerns the timestamp's temporal accuracy concerning the app's actual position within the historical timeline. As the ground-truth temporal location is hardly achievable in the vast majority of cases (i.e., it is not possible to know with absolute certainty when the sample was released), the main aim of the *timestamping* approaches is to provide a *good approximation* to the explored phenomenon in the absence of ground-truth data. In our case, a good approximation would minimize the amount of error for the majority of the samples and enable concept drift handling effectively. In this regard, due to the absence of a ground-truth temporal reference, our assumption is that a concept drift effective handling solution may provide relatively more stable and smoother performance over time when using an accurate timestamp than with an inaccurate timestamp, as data evolution may, in general, occur by shifting gradually over time introducing new elements and discarding others in a relatively smooth transition (i.e., *gradual* or *incremental* concept drifts). *Sudden* concept drifts may happen over time but their prevalence should not be significant (e.g., completely new malware outbreaks); otherwise, concept drift could be hardly modeled, and keeping high performance over time would be an intractable task.

10.2.1 A deep comparison of timestamps

The sequential steps and metrics used for the comparative analysis of the timestamps are described as follows:

1. ***Prevalence***: the prevalence of timestamps is a term related to *availability* which informs about the accessibility of the timestamp, that is, if the timestamp can be successfully collected or retrieved from the samples. For each timestamp, the number of *set* timestamps (i.e., properly defined) and *not set* timestamps (i.e., *missing* or undefined) for the whole data set was retrieved.

2. ***Validity***: the validity of a timestamp is an indicator of whether the timestamp is comprised in the Android historical time frame (i.e., from the 22nd of October 2008 [33] to the present day). It is not an indicator of accuracy; however, it discards all the timestamps not comprised in the *valid* Android timeline range (i.e., before 22nd of October 2008 or in the future).

3. ***Suitability***: the suitability of a timestamp combines the previous approaches in a positive way. Thus, a *suitable* timestamp is *available/prevalent* and is also *valid*. Consequently, a *non-suitable* timestamp is attributed to a timestamp that is *available* but *invalid*.

4. ***Distribution and statistical analysis***: data distributions for each timestamp and for each class were analyzed and compared using statistical measures. Histograms were used to visualize the data distributions and as input to statistical tests and techniques for similarity assessment. Two statistical methods for measuring the similarity between data distributions were used:

- ***Jensen–Shannon distance***: a distance metric that is calculated as the square root of the *Jensen–Shannon divergence*. The Jensen–Shannon divergence (JSD) is computed as:

$$JSD(P||Q) = \frac{D_{KL}(P||M)}{2} + \frac{D_{KL}(Q||M)}{2}$$

where P and Q refer to two probability distributions, M is the point-wise mean of P and Q calculated as $\frac{1}{2}(P+Q)$ and D_{KL} refers to the Kullback-Leibler (KL) divergence calculated for each pair of distributions. The KL divergence or *relative entropy*, which is used to quantify the difference between two probability distributions, is calculated as:

$$D_{KL}(P||Q) = \sum_{x \in X} P(x) \log\left(\frac{P(x)}{Q(x)}\right)$$

Based on these definitions, the distance metric provided by the square root of the JSD enables us to measure the similarity between two probability distributions. The JSD distance for two probability distributions is bounded in [0, 1] when the base 2 logarithm is used for computations. The general interpretation is that the larger the value (i.e., closer to one) the greater the difference between the distributions.

- ***Kolmogorov-Smirnov two-sample test***: a non-parametric statistical hypothesis test to assess the equality of one-dimensional probability distributions. It enables us to assess the probability that two collections of samples (i.e., $F(x)$ and $G(x)$) could have been drawn from the same probability distribution, that is, if they are statistically similar. The null hypothesis H_0 for the test is that the two distributions are identical (i.e., $F(x) = G(x), \forall x$), whereas the alternative hypothesis H_1 is that they are not identical. The Kolmogorov-Smirnov (KS) test answers the hypothesis by analyzing the maximum difference between the two experimental cumulative frequency distribution functions. The KS statistic is calculated as:

$$D_{m,n} = \sup_x \mid F_n(x) - G_m(x) \mid$$

where $F_n(x)$ and $G_m(x)$ refer to the empirical distribution functions of the two data samples, of size *m* and *n* respectively, and sup is the *supremum* function. For large samples, the null hypothesis is rejected at significance level α if

$$D_{m,n} > c(\alpha)\sqrt{\frac{m+n}{mn}}$$

where m and n are the sizes of the distributions and the value of $c(\alpha)$ is a parameter calculated as $c(\alpha) = \sqrt{-ln(\frac{\alpha}{2})\frac{1}{2}}$.

The similarity of distinct combinations of data distributions based on the timestamps was analyzed using the described metrics. As these metrics evaluate *similarity of distributions* using different approaches, the usage of both techniques provides a better overall perspective of the differences between the analyzed sets.

5. ***Accuracy***: an approximation of the accuracy of the timestamps was explored to assess their reliability. The evaluation of timestamp accuracy is a significant challenge due to the absence of an exact ground-truth timestamp. For this purpose, open-source intelligence feeds such as malware family discovery news of specific malware

families by antivirus vendors and media sources were used to establish an approximation of the discovery date of a specific malware family. After that, a time frame around the date was established (i.e., ± 6 months) and statistics were retrieved from each timestamp data distribution for each family. The rationale is that, if the timestamp is accurate, it would place the samples around that time frame (i.e., discovery time ± 6 months) and also after it, implying that the malware family might be located accurately and also its evolution. If a significant number of samples is placed outside of this time frame, the timestamp could be deemed relatively inaccurate. Despite the limitations of this approach, the experimental results proved that it provides a good notion of the accuracy of the timestamps, especially when the results among timestamps are compared.

10.2.2 Experimental results

It is worth noticing that from the set of six timestamps analyzed, two correspond to *external* timestamps, set in this case by VirusTotal scanning reports, thus not extracted from the *apk* archive metadata. These timestamps are the *first seen* and the *first seen in the wild*. An external timestamp is less prone to be manipulated by perpetrators as it is not in the immediate scope of the attacker. However, they can be prone to delays, as they depend on users' proactive behavior (i.e., user submission to VirusTotal's service), and processing errors. Besides, the *first seen in the wild*, defined as the first time the app was observed anywhere across the internet, might be not set for benign applications. The remaining four timestamps are *internal* timestamps, collected from the inner files of the app archive data. Thus they can be manipulated or removed by a motivated attacker.

The following sections describe and provide a comparative analysis of the timestamps from different perspectives in relation to *availability* and *reliability* measures.

1. ***Prevalence***: provides a notion of the data availability, which is critical to build effective learning systems. If the timestamp cannot be retrieved, the sample cannot be located in the historical timeline and, consequently, the sample is unusable. Figure 34 conveys graphically the *prevalence* or *availability* property for each timestamp in the whole data set. The horizontal axis provides the timestamps, referenced in their abbreviated form. For each timestamp, two vertical bars are defined, which report the relative frequency or *percentage* of data samples that had the timestamp *available*. The colored areas refer to class-wise proportions (i.e., red for malware, green for benign apps), while the grey areas indicate the proportion of data samples that did not have the specific timestamp available.

   As can be observed from Figure 34, the majority of the timestamps are available for all data samples, as most of the bars reach beyond 97% prevalence. Furthermore, the *first seen* was defined for all data samples. This was expected as to retrieve the report for the first time the timestamp is always set by the scanning service. The internal timestamps are mostly available, especially for malware instances. Interestingly, a larger proportion of null-valued timestamps was found on legitimate apps, a fact that could appear counter-intuitive, but that was also highlighted by [27]. An exception to the high availability of the timestamps is the *first seen in the wild* timestamp. The majority of the reports retrieved did not provide information for this feature, thus it is missing for most of the apps, especially for benign apps (i.e., 0.6%). This is logical as the objective of the scanning service is to detect malware threats (i.e., positive detection) thus the *first seen in the wild* location for benign instances is not a priority as it is usually irrelevant.

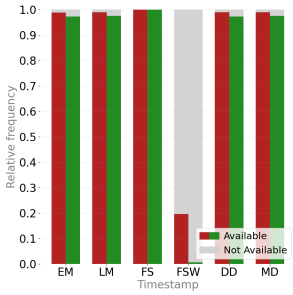


Figure 34: Availability [37]

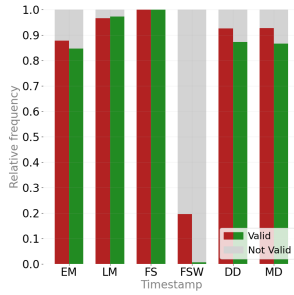


Figure 35: Validity [37]

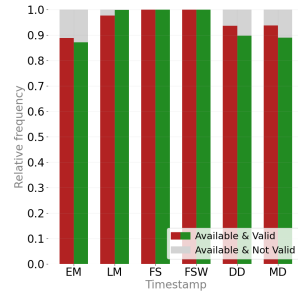


Figure 36: Suitability [37]

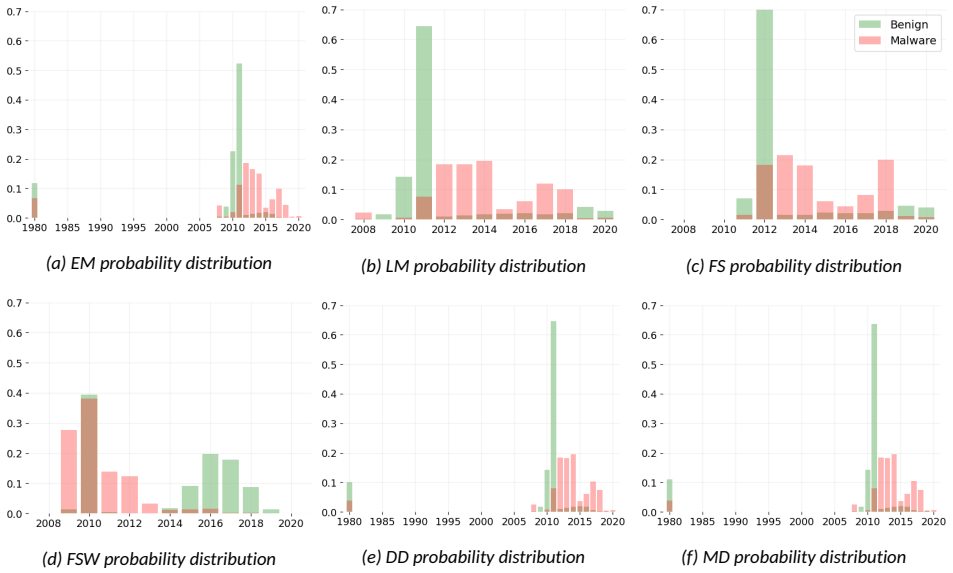
2. ***Validity***: The timestamp for any Android app sample should be located within the Android historical timeline, which encompasses from the 22nd of October 2008 (i.e., Android Google Market public release) to the present day. A timestamp located within this time frame is deemed as *valid*. Timestamps located in the future (e.g., 2107) or in the past (e.g., 1997), which were found within the data, are impossible configurations, suggesting tampering, and were consequently labeled as *not valid*. Figure 35 reports the *validity* property for each timestamp in the whole data set. Similarly to Figure 34, the horizontal axis provides the timestamps, referenced in abbreviated form. The vertical bars report the proportion of *valid* timestamps for each class with green/red color and the *not valid* as shaded areas.

   Figure 35 reports similar values to the ones in Figure 34 for *FS* and *FSW* timestamps. However, for the *EM*, *LM*, *DD* and *MD* timestamps, the bars reach lower figures, especially for the *EM* timestamp. This indicates that this timestamp is the one that contains more non-valid values, followed by *DD* and *MD* timestamps. In all cases, with the exception of *EM*, malware samples reach higher values than benign samples, which, again, seems counter-intuitive. However, this fact may only reflect a general disregard for timestamps by benign app developers but does not provide any hint about the accuracy of the timestamp.

3. ***Suitability***: in this analysis, the concept of *suitability* provides a notion about the most *usable* timestamps, that is, they are *available* and *valid*. Figure 36 reports the *suitability* proportion per timestamp and class. In this case, the colored areas refer to class-wise timestamps that are both available and valid. The grey areas report the proportion of samples that have available but invalid timestamps.

   Figure 36 conveys that, *FS*, *FSW* and *LM* are the most *suitable* timestamps, with a large proportion (i.e., 100% for *FS* and *FSW*) of *available* data that lie inside the *valid* time frame. However, despite the high *suitability* of *FSW*, its low prevalence makes it a worse option than *FS* and *LM* if data quantity is a requirement. *EM*, *DD* and *MD* timestamps show values ranging from 87% to 93% thus deemed as the least *suitable* options.

4. ***Distribution and statistical analysis***: The probability distribution of each individual timestamp is provided in Figure 37. For each graph, the color of the bars refers to the class distribution (i.e., green for benign software, and red for malware). The X-axis provides the year range of the bar data, while the Y-axis provides the relative frequency for each year. The horizontal range is adjusted to the valid time frame for those timestamps that only provided valid data and the *LM* timestamp. This was



(a) EM probability distribution (b) LM probability distribution (c) FS probability distribution

(d) FSW probability distribution (e) DD probability distribution (f) MD probability distribution

Figure 37: Probability distribution for each timestamp [37]

preferred given the negligible proportion of invalid values of this timestamp and also to provide an enhanced visual comparison between *LM* and *FS* (i.e., the most *suitable* timestamps) which is enabled when just the *valid* range is plotted.

As can be observed in the graphs of Figure 37, the *internal* timestamps (i.e., *EM*, *LM*, *DD*, *MD*), show similar data distributions. The *LM*, however, does not show the large proportion of data points (i.e., around 10%) located in 1980 that the other *external* distributions have, but the distributions in the valid range are relatively similar, especially when compared with *DD*, *MD*. The *FSW* data distribution is radically different to the other distributions, showing malware data concentrated in the 2008–2016 range and legitimate data in the 2014-2019 range and also in 2010. When *LM* and *FS* are compared, the two distributions seem relatively similar for legitimate data, peaking in one year and relatively low in the other years. However, the peak occur in 2011 for *LM* and 2012 for *FS*. In the case of malware, the three consecutive bars around 0.2 probability value occur in both distributions in the range 2012–2014. Despite that, the distribution of data surrounding the 2012–2014 time frame is different, with many samples in the years before this time frame for the *LM* but not for the *FS*, which concentrates most of the samples in the years after this range, especially in 2018. These observations may suggest that the relatively similar but shifted shapes might have been caused by recurring *delay* in *FS* with respect to *LM*. Further exploration of this hypothesis is addressed in Section 10.3.

The statistical analysis of timestamps distributions enables the assessment of their similarity, which provides a notion of the degree of variability among them. For this purpose, *Jensen-Shannon distance* (i.e., JS) and *Kolmogorov-Smirnov 2-sample test* (i.e., KS) were used. The former uses the notion of distance between distributions to provide a similarity score, bounded in the [0, 1] interval, where higher values report greater dissimilarity, while the latter uses the concept of *p-value* to assess the statistical significance of the results by accepting or rejecting the null hypothesis

(i.e., the distributions are equal) at a specific confidence level α. Thus enabling the assessment of the similarity between the distributions. Small *p-values* indicate a high probability that the distributions come from the same population, suggesting a greater similarity between the timestamps.

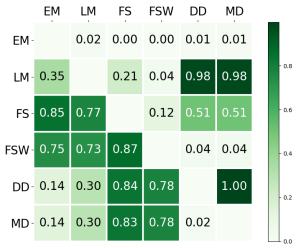


Figure 38: JSD-KS matrix for benign data [37]

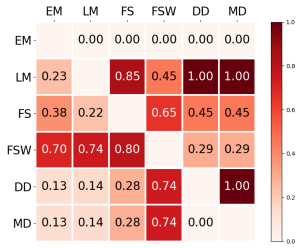


Figure 39: JSD-KS matrix for malware data [37]

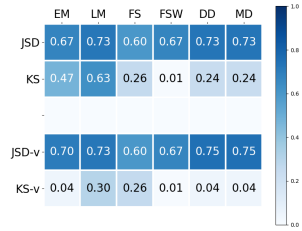


Figure 40: JSD-KS matrix for inter-class data [37]

The matrix in Figure 38 provides the comparison among all pairs of timestamp distributions for the benign data. Given the symmetry of the calculated measures, they enable us to provide the computed values for both similarity measures in the form of a matrix where the main diagonal is left blank to separate the values. The values above the diagonal of the matrix provide the values for *KS* computations, while the values below the diagonal provide the obtained *JSD* values.

As can be observed in Figure 38, all the timestamps seem to provide distinct distributions for the samples. The only exception is for *DD* and *MD* which have an almost 0 distance (i.e., almost perfect similarity) and a *p-value* of 1. These values indicate that these two distributions are roughly the same. A fact that was also spotted in the graphical visualization in Figure 37. Further, *DD* and *MD* have a high degree of similarity (i.e., they have small distance and high KS value) with *LM* and in less degree with *EM*. This fact shows the relatively close distance between the distributions based on *internal* timestamps, which also confirms the spotted similarities on the plots. Despite that, *LM* is preferred as it provided more *suitable* timestamps than *DD* and *MD*. Regarding the *external* timestamps, *FS* and *FSW*, they show unique distributions, as evidenced by the large distance values and small p-values with almost all the other distributions. When the most *suitable* timestamps are compared, *FS* and *LM*, they are seen to have large dissimilarity (i.e., 0.77; therefore, they locate the samples distinctly) but relatively low p-value (i.e., 0.21), which confirms the interesting observation from the graphs. Their cumulative functions are close enough to be found relatively similar using the KS-test but different when the distance-based similarity is used.

The matrix in Figure 39 provides the comparison between all pairs of timestamp distributions for the malware data. Again, the area above the diagonal provides KS *p-values*, while the area below the diagonal provides the JSD values.

As can be seen in Figure 39, the overall picture is similar to the benign case. The *internal* timestamps have high similarity. The *external* timestamps differ significantly, more than in the benign case. An exception to the overall similar observations to the benign case emerges when comparing *FS* and *LM*. For the malware case, these two distributions have a smaller distance but a significantly higher p-value. This fact shows that the range and overall shape of the distributions are similar but that the cumulative function of the values is significantly distinct. These results support the

hypothesis of the *delay* between them, and the greater concentration of data in the early years for *LM* and for the latter years of the valid range for *FS*.

The previous statistical analysis compared the distribution of data according to timestamps for the same class (i.e., benign and malware). An interesting comparison is also the analysis of the similarity of class distributions within a given timestamp. The results of this comparison are reported in Figure 40, where the columns provide information about the specific timestamp and the horizontal rows about the statistical value computed when comparing the benign and malware distribution for each specific timestamp. The upper rows in the figure provide the comparative results of JSD and KS for the whole distributions, whereas the lower rows provide the same information but just for the distributions in the valid time frame, indicated as *JSD-v* and *KS-v*.

The overall interpretation of Figure 40 is that the valid time frame emphasizes the differences among class distributions. The values of distance increase and *p-values* diminish in the lower rows (i.e., valid range) when compared with the upper rows (i.e., whole range). The only exception to this are *FS* and *FSW* timestamps which have the same values on both pairs of rows as they are always *valid*. Therefore, the class-based distributions are significantly different across all timestamps.

5. ***Accuracy***: due to the absence of ground-truth timestamps, the assessment of timestamp accuracy and reliability is hindered and can only be approximated using *open source intelligence* sources which might be relatively delayed and not fully precise. In this research, the first publicly available report for specific malware families was used to approximately contextualize the malware family within the historical timeline. Table 14 shows 10 malware families (i.e., one per row) and the date used as a reference of the discovery time frame based on reports taken from reliable sources and contrasted with other sources (i.e., month/year). The data sources are provided in square brackets. The following six columns are split into three sub-columns which are referred as ±6, > 6, and NV. For the sake of interpretation of the table, these columns have been colored in green, yellow, and grey, respectively. These columns provide the proportion of data samples (i.e., percentages) of the data set, dated with each specific timestamp, that *lie within the reference value ±6 months* (i.e., ±6 column), *beyond the reference value + 6 months* (i.e., > 6 column), or that have a *invalid* location (i.e., NV column). As mentioned before, a precise timestamp should locate most of the samples of a specific malware family between the ±6 and > 6 range. The proportion of NV values and samples located *before* the valid range should be minimal. A higher proportion of values within the > 6 range may imply a delay in the timestamp or family evolution. The ±6 range gives a notion of the number of samples within this range but due to family evolution, it can only be interpreted in comparison with the other values, as the data set may contain fewer *original* samples than evolved samples. Furthermore, the malware family naming is inconsistent among AV vendors or even analysts thus being a handicap for any malware family analysis. In our study, we assume that most of the labels are certain which provides a relative degree of flexibility to interpret the results. The last row of the table provides the average value of each column for each specific timestamp. The difference between the sum of the three values (i.e., ±6, > 6, and NV) and 100 is attributed to the proportion of timestamps dating the sample in the valid range but before the reference value - 6 months.

   As can be observed in Table 14, the individual proportions for specific malware fam-

ilies greatly vary among timestamps. A better general picture is provided by the total values. Even though *outlier* values may cause the average to be skewed, it still serves as a reliable indicator of the general trend. The *LM* timestamp offers the best suitable accuracy characteristics according to the interpretation of the *total* values. It has a very low ratio of invalid values and a high proportion of timestamps within the *valid* time frame. The average values also confirm that the internal timestamps show similar statistics/distributions, with all of them showing similar proportions but with significantly lesser *invalid* values for the *LM* timestamp. The external timestamps show completely different pictures. The *FS* is characterized by providing always valid values, whereas the *FSW* shows a large proportion of invalid, which correspond to *missing* data in this case. Finally, when the *FS* and *LM* timestamps are compared, the average values show that *FS* captures most of the data beyond the reference + 6 months (i.e., > 6) timestamp, whereas the *LM* timestamp does it in similar proportions on both valid ranges. This supports the delayed nature of *FS* to capture malware outbreaks and the goodness of *LM* to locate most of the data samples with improved precision.

Table 14: Accuracy of timestamps for malware families [37]

Family	Reference	EM			LM			FS			FSIW			DD			MD		
		±6	>6	NV	±6	>6	NV	±6	>6	NV	±6	>6	NV	±6	>6	NV	±6	>6	NV
Geinimi	11/10 [30]	31.3	43.8	6.3	56.3	43.7	0	12.5	87.5	0	18.8	0	56.3	56.3	43.7	0	56.3	43.7	0
DroidDream	03/11 [29]	65.9	19.8	0	67	28.6	0	14.3	85.7	0	16.5	4.4	61.5	67	28.6	0	67	28.6	0
DroidKungFu	06/11 [56]	66.2	13.1	11.9	68	31.6	0.3	7.6	92.4	0	2.5	16.5	54	68.3	31.4	0.3	68.3	31.4	0.3
Plankton	06/11 [91]	23.8	71.4	1.6	22.2	77.8	0	6.3	93.7	0	0	1.6	92.1	22.2	77.8	0	22.2	77.8	0
GinMaster	08/11 [109]	21.2	70.4	6.6	17.6	81.5	0.2	0.7	98.9	0	5.2	3.9	69.8	18.4	80.7	0.2	18.4	80.7	0.2
AnserverBot	09/11 [56]	96.3	0	1.0	99.7	0	0	42.1	57.9	0	0.3	47.2	39.1	99.7	0	0	99.7	0	0
Slocker	05/14 [69]	23.2	50.3	25.8	23.1	57.9	18.4	1.1	98.5	0	0.1	1.1	93.9	23.2	55.1	21.1	23.3	54.4	21.7
MobiDash	01/15 [62]	3.3	60.1	36.7	3.3	90.2	6.5	0.7	99.3	0	0.7	1.3	96.7	3.3	60.8	35.9	3.3	60.1	36.6
BankBot	01/16 [26]	60.4	17.3	12.3	60.5	20.4	9.2	71.3	27	0	4	0.2	76	60.5	20.2	9.4	60.5	19.6	10
Triada	03/16 [19]	41.7	0	58.3	41.7	16.7	41.6	41.7	58.3	0	41.7	8.3	50	41.7	16.7	41.6	41.7	16.7	41.6
Total	-	43.3	34.6	16.1	45.9	44.9	7.6	19.8	80	0	9	8.5	69	46	41.5	10.9	46	41.3	11

10.3 *Last modification* vs. *first seen*: a comparative analysis

The results from Table 14 and the previous analysis suggest that *FS* and *LM* are significantly better timestamps than the other analyzed approaches in terms of suitability, statistical properties, and accuracy. To explore their relation deeply, the next paragraphs analyze statistically the differences between them and their *delayed* relation over time.

Figure 41 and Figure 42 provide the differences between both timestamps, computed for each data sample, separately for benign and malware data. The base unit is days and the base timestamp used is the *last modification*, so that the differences can be expressed in positive terms (e.g., +8 days). The assumption for this is that the last modification timestamp would place the sample more accurately in the historical timeline, earlier in time with respect to the first seen timestamp. Therefore, it was chosen as the reference time. These graphs report relevant descriptive statistics regarding the temporal differences (i.e., Y-axis) for the samples in the data set, located in a specific time period (i.e., six months chunks) using the last modification timestamp (i.e., X-axis), with respect to the first seen timestamp. The data are split into chunks of six months data (i.e., period) for better interpretation of the results and deeper exploration of the differences. Just the valid time frame is plotted (i.e., from 2009 to 2020). The semesters are referenced as appended suffixes to the year figure (e.g., 2009.1 reflects the first six months of 2009). The differences were calculated individually per data sample and grouped into data periods based on the last modification timestamp. Descriptive statistics are calculated per period: mean, median, minimum difference, and standard deviation. The blue solid line reports the average

value for each period, while the blue dashed line provides the median. These two central tendency measures report the average value of displacement of a sample for each period. The green line provides the minimum difference value found in that period. The standard deviation, plotted as a blue area, conveys the average spread of the differences around the mean for the data samples located in each specific chunk.

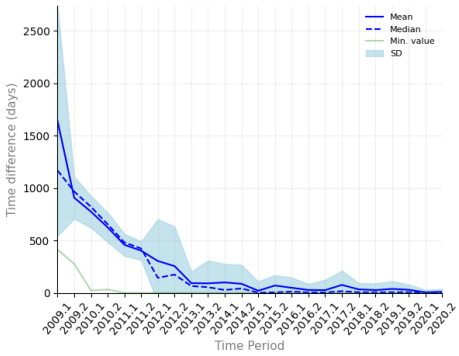


Figure 41: Differences between LM-FS *timestamps for benign data [37]*

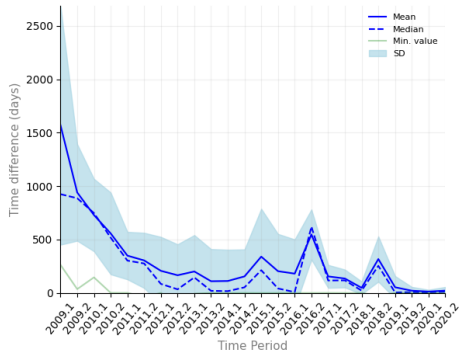


Figure 42: Differences between LM-FS *timestamps for malware data [37]*

For both classes, a positive difference between both timestamps is observed. This evidences that the *first seen timestamp* locates the samples later in time, thus *delayed*, with respect to the *last modification* timestamp. This fact is especially pronounced in the early years of Android history, with average differences of around 1,500 days (i.e., four years) for both malware and benign applications. This means that an instance located in 2009.1 (i.e., the first semester of 2009) by the last modification timestamp might be located by the first seen timestamp in 2013.1 (i.e., the first semester of 2013). This significant difference might impact the performance and adaptation capabilities of a trained classifier to deal with concept drift as the *first seen* may generate *artificial drift* by misplacing the data, which might be more complex to model effectively than real concept drift, generally smoother.

However, as can be observed in Figure 41 and Figure 42, the differences between these timestamps have been reducing over time, as evidenced by the monotonically decreasing mean value for benign instances and the significant decrease occurred in the case of malware instances, especially for the recent years. This fact makes the timestamps more synchronized and closer over time, even equivalent for the 2019–2020 time frame. For instance, 2020.1 and 2020.2 periods have mean values of 4.88 and 12.37 days and median values of 2 and 3 days, respectively, for benign samples, and average values of 15.9 and 16.45 days and medians of 5 and 11 days, respectively, in the case of malware samples. This is a dramatic change when compared to 2010.1 statistics, which show a mean value of 728.4 days and a median of 747 days for malware and an average of 774.3 days, and a median of 821 days for benign data. As a result, the gap between both timestamps to date samples has significantly decreased over time making them converge and, consequently, increasing the reliability and accuracy of the *first seen* timestamp in the recent years (i.e., 2019–2020).

The convergence of both timestamps supports the hypothesis that the *last modification* timestamp is accurate and that is *rarely* tampered by attackers. Consequently, if the system has to learn from past data and predict past samples, it might be safer to use the *last modification*, whereas if the system uses mainly recent data, the convergence of the

timestamps implies that both should be appropriate and perform relatively well against data drift. Furthermore, if data tampering is a major concern, the usage of the *first seen* ensures that the data have not been tampered with, even though a minimal delay should always be assumed.

10.4 Timestamp performance analysis for concept drift handling

The main objective of the experimental scenario was to evaluate the goodness of the analyzed timestamps to deal with concept drift when the detection model is induced in different feature spaces. The concept drift handling method proposed in Section 6.3.1 was used as a continuous detection model. Due to the characteristics of *KronoDroid* data and the demands of machine learning models, the experimental setup was restricted to the period encompassing the second semester of 2011 until the first semester of 2018. This time frame spans seven years of Android history, including the most active years in Android malware development [13, 57].

The available data per timestamp for the selected time period (i.e., from 2011.2 to 2018.1) are provided in Table 15.

Table 15: Sample size per timestamp from 2011.2 to 2018.1 [37]

Timestamp	Malware	Benign	Total
EM	33,346	7,602	40,948
LM	38,496	13,456	51,952
FS	40,376	32,870	73,246
FSW	2,137	116	2,253
DD	36,805	11,555	48,360
MD	36,810	11,500	48,310

As can be noticed in Table 15, *FS* provides most of its data within this range, whereas the external timestamps provide a smaller number of samples. As expected, the data is imbalanced towards the malware data, thus justifying the usage of a data set balancing technique to avoid any class bias from the classifier. Finally, as the data provided by the *FSW* timestamp is not enough to build a single classifier, this timestamp was discarded and not used in the following experimental setups.

Three experimental scenarios were induced to explore the different feature spaces separately using the timestamps. The F_1 score performance metric was retrieved. The graphs provided in the following sections report each model's performance with different colors and/or line styles. More precisely, *EM*, *DD* and *MD* are provided in green color and with solid, dashed, and dotted line style, respectively. *LM* is reported with a blue solid line, while *FS* with a red solid line. As the most *suitable* options were *LM* and *FS*, their average performance for the whole analyzed time frame is reported with a horizontal line, in solid blue color for *LM* and in solid red color for *FS*.

10.4.1 Permissions feature space

Figure 43 shows the performance outcomes of the models created for the permissions feature space and the unique timestamps. The permissions feature space is categorical and the smallest of the analyzed ones. As can be observed, despite suffering two sudden drops (and consequent recovery), the smoothest line is provided by the *LM* timestamp. The other internal timestamps provide similar performance but describe a rougher surface. The *FS* timestamp performance is not smooth, characterized by several sudden peaks and bottoms in neighboring quarters, especially at the beginning and the end of the analyzed time frame (i.e., bumpy red line). Furthermore, it has the lowest average perfor-

mance in the analyzed period. Despite that, the overall performance of all models is over 0.90 F_1, which reflects the goodness of the system to deal with concept drift, especially the natural data drift, which is better captured by the *internal* timestamps.

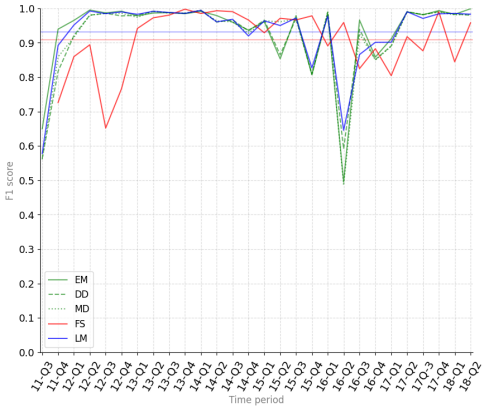


Figure 43: Timestamps F_1 performance on the permissions feature space [37]

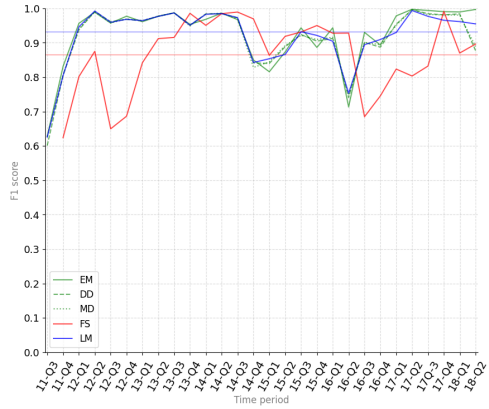


Figure 44: Timestamps F_1 performance on the system calls feature space [37]

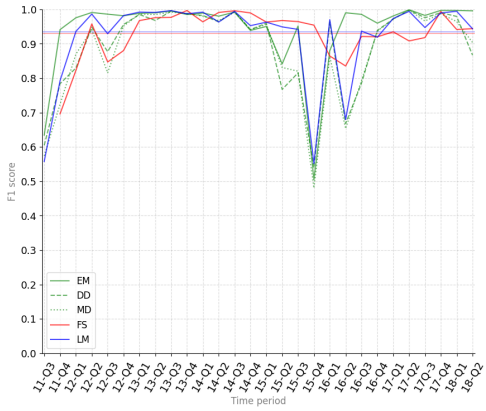


Figure 45: Timestamps F_1 performance on the API calls feature space [37]

10.4.2 System calls feature space

Figure 44 shows the performance of the models built for the system calls feature space. The system calls feature space is numeric and larger than the permissions space. Similar to Figure 43, the internal timestamps provide smoother performance lines, and, again, the *LM* timestamp seems to provide the best performance. It enables the model to capture better the natural data drift, showing steady recovery after sudden data drifts. However, in this case, *EM* achieves similar performance over the whole range but shows a more *noisy* performance line. As in the permissions case, *FS* provides the worst performance and is characterized by sudden dips and peaks, which are likely to be caused by the *artificial* data drift. An interesting fact in this plot is that from 13-Q3 to 16-Q3, *LM* and *FS* seem to perform synchronously. The *FS* performance line is relatively similar but delayed

one quarter with respect to *LM* and reaches more extreme values. This goes in line with the median differences observed in this time frame in Figure 41 and Figure 42 (i.e., below 90 days for both classes). Lastly, in this case, the difference between *FS* and *LM* average performance is significantly greater, with the average performance of *LM* around 0.93 and at the 0.87 level for *FS*.

10.4.3 API calls feature space

The performance of the models built for the API calls feature space are provided in Figure 45. The API calls feature space is the largest feature space, with over 53,523 features. Similar to the other feature spaces, the performance of the internal timestamps is similar and over the performance of *FS*. However, in this case, two large dips are observed for the internal timestamps that are not observed in the case of *FS*. It is worth noticing that, in the case of high-dimensional spaces, the quantity of data is critical to build effective models (i.e., data density is needed to build precise classification boundaries), a phenomenon called *curse of dimensionality*. As reflected in Table 15, the data sets available for the internal timestamps are smaller in general, and significantly reduced on these specific data periods; therefore, the reduced performance could be the result of insufficient data to cover the feature space and build an effective model. However, despite the large dips in those specific periods, the average performance of *LM* still outperforms *FS*.

10.5 Chapter summary

Only a tiny proportion of the research regarding Android malware detection has considered concept drift in their design or experimental setup. In this regard, timestamps, a central element for concept drift handling, have not received the needed attention as none of the concept drift-related studies considered more than a single timestamp in their proposal. This chapter presented an extensive benchmarking about timestamp options and their capabilities to deal with concept drift effectively in distinct feature spaces. The results show that timestamp selection is a critical decision and that the *last modification* and *first seen* timestamps are the best options to build effective long-lasting ML models for Android malware detection under data evolution constraints.

11 Applying active learning to handle data evolution in Android malware detection

The methodology proposed in Chapter 6 enables concept drift handling using a data stream approach and a pool of classifiers to update and refine the knowledge of the detection system over time. Despite the remarkable benefit of the described approach and others in the related literature, *model retraining* is the most common approach to handling concept drift. This simple approach proposes the update of the model by retraining it with a new set of data when drifting is observed (e.g., a drift detector is used to signal concept drift) or periodically to keep the model knowledge updated with recent data samples. In either case, most of the approaches in the literature assume that the ground-truth label is available [71]. This implies that the data must be labeled by experts prior to their usage to update the model. In this regard, data labeling can become expensive and time-consuming when the process demands significant expert knowledge to be performed (e.g., malware analysts). Additionally, according to statistics, thousands of Android malware samples are found each month, necessitating enormous effort from companies and teams of malware experts to evaluate all the incoming samples [14].

The active learning approach can be used to minimize data labeling efforts while aiming for high-performance metrics. In our study, we investigated the suitability of the active learning approach to minimize the data labeling cost in those environments where a wealth of unlabeled data is available, and its usefulness to enhance model retraining in a non-stationary data environment. The tested scenarios simulate the data processing and adaptation to concept drift needed in the case of a malware scanner company/service where a wealth of unlabeled data is available and the necessary skills for proper labeling make the analysis of the whole data received costly and infeasible.

The work explained in this chapter is based on Publication X which is currently undergoing a peer-review process. This work has been included in the thesis to support the related research and make a more complete narrative of the whole research performed.

11.1 A brief on active learning

Active learning is a form of *semi-supervised learning* based on the assumption that an ML algorithm can yield better performance with fewer training iterations (i.e., less data) if it is allowed to select the data from which it learns [88]. In the usual active learning scenarios, a supervised model, trained with a small quantity of labeled data, is allowed to request the labeling of specific instances from a collection of unlabeled data samples by an *oracle* (i.e., human expert). The main objective of the active learning approach is to achieve high-performance models using as few labeled samples as possible, thereby minimizing the cost of the data labeling process.

The selection of the specific instance for labeling (i.e., query instance) at each training iteration is based on an *informativeness* assessment of the whole set of unlabeled instances performed by the active learner using a query strategy [89]. The pool-based framework, depicted in Figure 46, is the most common active learning approach. This approach assumes the existence of a small set of labeled data, which is used to induce the initial model, and the availability of a large *pool* of unlabeled samples [88]. Query instances are selected by the classifier itself from the unlabeled pool for expert annotation. Once annotated, the labeled data are included in the labeled training set which is used to update the knowledge of the supervised model.

Different query strategies have been developed to select the *most informative* instance from the whole set of unlabeled instances [89]. The most commonly used ap-

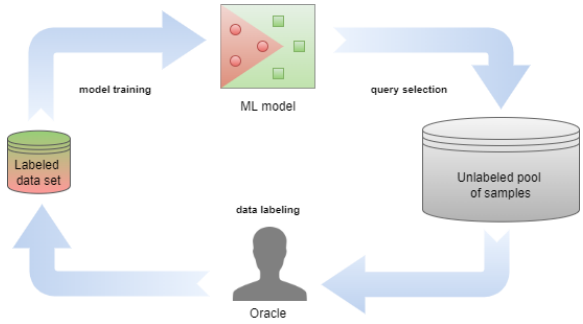


Figure 46: Pool-based active learning framework

proach is *uncertainty sampling*, where the query instance is selected based on the assessment of how certain is the learner about the class of the unlabeled samples. In the *classification uncertainty* scoring strategy, the learner selects for labeling the instance (x) for which it is least certain about how to label (i.e., greatest uncertainty). The *classification uncertainty* metric is based on a *least confidence* score (U) computed as:

$$U(x) = 1 - P(y^*|x) \tag{4}$$

where x is a specific instance and y^* is the most likely prediction for that instance.

11.2 Testing scenarios

The data set used for the experimental tests was the *real device KronoDroid* data set. The *first seen* timestamp was used to order the data samples along the historical timeline and simulate a realistic scenario of the existence of an incoming data stream. We were able to pose as a malware scanner firm dealing with an Android malware detection system that was prone to concept drift issues by using this timestamp to replicate the continuous flow of user-submitted data samples to a scanning service.

For detection model induction, three sets of input features were used to describe the apps, namely, static (permissions), dynamic (system calls), and hybrid (system calls and permissions) with lengths 166, 288, and 454, respectively. The same initial classification model was used in all the experimental scenarios. This initial model was a Random Forest instance trained with data belonging to July and August 2011. This initial time frame was selected as it provided enough data to generate a high-performance initial detection model. However, as this initial training data set was not balanced, a data balancing technique was applied to generate a balanced training set for initial model training. In this regard, two data balancing methods were used (i.e., *random undersampling* and *random oversampling*) and their impact was evaluated. The remaining ordered data samples were split into consecutive data chunks using temporal and data set size constraints. Based on experimental tests, the maximum temporal constraint or time window was set at 60 days and the maximum data pool size set to 4,000 (i.e., unlabeled data). The time period analyzed encompasses seven years of the Android history, from the initial time frame (i.e., July–August 2011) to May–June 2018.

In the testing scenarios, the initial classifier was retrained using different concept drift-handling strategies. The strategies evaluated to handle concept drift using model retraining are described as follows.

- *Batch retraining*: this strategy updates the detection model by retraining the classi-

fier using the whole amount of data available in each specific chunk. The retrained model is used to forecast the labels for each subsequent period. Therefore, at time t all data from previous time periods (i.e., $s_0, .., s_{t-1}$, where s identifies a data set belonging to a specific time period t) was used to update the model, and forecast the labels of s_{t+1} data. Next, the whole data set belonging to $t+1$ was used to update the model and forecast labels for s_{t+2}. This cycle was repeated for each data chunk until the end of the analysis period. This batch retraining approach is the frequent solution used for concept drift adaptation and was used as a baseline in our experimentation.

- *Active learning*: this strategy updated the detection model by selecting the *most informative* instances for each data chunk, one at a time until a predefined performance threshold was reached (i.e., $0.95 \geq F_1$). The *classification uncertainty* score was used to rank and select one instance at a time from the unlabeled pool of instances (i.e., whole data chunk). The selected instance was labeled by the *oracle* and used to retrain the model. The rest of the data chunk was used to evaluate the performance increase/decrease after the single retraining step. The training cycle, as depicted in Figure 46, was repeated until the threshold of 0.95 F_1 was achieved. The remaining data, not used in the iterative training steps were discarded and the trained model, as in *batch retraining*, was used to forecast all the samples for the next period. If the performance retrieved processing all the data chunk was lower than the established threshold, the model was rolled back to its best performer configuration and used to forecast the data from the following period.

- *Random sample selection retraining*: this strategy uses the same iterative training steps as the active learning approach but, in this case, no score is used to select the most informative instances from the unlabeled pool of samples. Random sample selection was utilized instead. We were able to recreate the scenario where a large amount of unlabeled data was available, but no special criteria were utilized to choose the examples; thus, they were selected randomly. This model offers a baseline for evaluating the sample selection method's efficacy in minimizing data labeling.

The performance of the induced models using the different sets of features for all the strategies was retrieved and compared. In all cases, the model trained using data from period t was used to forecast the labels of the data belonging to the subsequent period, s_{t+1}. The main difference among the approaches lies in the strategy used to select the samples for model updating to handle concept drift (i.e., all data, random selection, or uncertainty score).

The performance of the detection models using the described retraining strategies to handle concept drift was evaluated using two relevant binary classification performance metrics: *accuracy* and F_1 *score* metrics. Given the randomness of the sampling techniques and the base algorithm used, all the scenarios were repeated 30 times.

11.3 Experimental results

Table 16 provides the obtained results using all the described concept drift-handling approaches. More specifically, the *feature set* column describes the input features used by each specific model tested and the *balancing method* reports the technique used to balance the initial data set, in the case of the two query strategies used (i.e., random and uncertainty), and in all data chunks for the batch approach (i.e., to avoid that the imbalanced

data chunks generated biased RF models). For each combination of the feature sets and balancing methods, three strategies to handle concept drift were used and referenced in the *query strategy* column. The remaining columns report the performance metrics that enabled us to analyze and compare all the approaches evaluated. The *labeled samples* column informs about the average number of samples processed by each model (i.e., , $\bar{x}$), that is, the number of instances labeled to reach the performance threshold in the case of the query strategies, $F_1 \geq 95\%$. The columns F_1 *score* and *accuracy* provide the average performance of the trained models in all time windows in the analyzed time frame (e.g., 45 data chunks spanning between September–October 2011 and May–June 2018). The reported values for labeled samples and the performance metrics are the average values of the 30 tests performed for each specific scenario. The standard deviation (i.e., s) is reported to better contextualize the mean value as a data descriptor. Additionally, for the *labeled samples*, the proportion of the average number of labeled samples reported in relation to the total data available in the analyzed period is reflected by the $\%$ column.

Table 16: Results of the testing scenarios [36]

Feature set	Balancing method	Query strategy	Labeled samples			F_1 score		Accuracy	
			$\bar{x}$	s	%	$\bar{x}$	s	$\bar{x}$	s
Permissions	Oversampling	Batch	67,068	0	100	91.2	0.4	92.5	0.4
		Random	30,100.4	129.8	44.9	89.4	1.0	90.3	1.0
		Uncertainty	11,845.6	41.7	17.7	89.4	0.6	90.4	0.6
	Undersampling	Batch	67,068	0	100	90.9	0.8	92.4	0.8
		Random	29,409.9	113.5	43.9	89.5	1.1	90.3	1.1
		Uncertainty	9,281.4	35.5	13.8	89.6	0.7	90.5	0.6
Syscalls	Oversampling	Batch	67,068	0	100	85.1	0.8	86.1	0.7
		Random	45,028.9	127.8	67.1	84.1	0.9	84.9	0.9
		Uncertainty	13,098.8	38.6	19.5	84.5	0.9	85.3	0.8
	Undersampling	Batch	67,068	0	100	82.7	1.2	83.3	1.2
		Random	45,378.7	118.5	67.7	84.5	0.9	85.0	1.0
		Uncertainty	12,748.3	52.3	19.0	85.1	1.0	85.5	1.0
Hybrid	Oversampling	Batch	67,068	0	100	92.8	0.5	93.5	0.4
		Random	22,057.2	121.5	32.9	90.9	1.0	91.2	1.0
		Uncertainty	1,991.9	8.9	3.0	91.6	1.2	91.9	1.2
	Undersampling	Batch	67,068	0	100	92.5	0.6	93.1	0.6
		Random	20,978.4	116.1	31.3	91.0	1.1	91.1	1.1
		Uncertainty	1,459.4	6.3	2.2	91.7	1.4	91.9	1.4

Table 16 displays that when the permissions feature set is used, the active learning approach provides similar performance as the baseline model (i.e., batch, using all data), but requires the smallest quantity of data among the tested strategies. More precisely, the uncertainty-based active learning approach minimizes the data labeling needs to achieve similar performance as the other two approaches, using either of the balancing techniques. The batch approach, which requires the labeling of all the data samples shows slightly better performance than the active learning and random approaches, but these show significantly lower data labeling requirements. In this regard, the uncertainty-based active learning approach outperforms the random selection approach by using less than 18% of the total data available in the analyzed time frame (i.e., 67,068 data samples), in both cases. Even though both single query-based retraining approaches show benefits over the batch approach, the active learning approach requires three times fewer data than the random instance selection to achieve the same performance metrics. This fact evidences that, in the permissions case, the single query-based gradual modification of the classifier decision boundary shows benefits when compared to the baseline model which uses batch processing, that is, all data. The random approach shows slightly lower performance than the baseline model, but with less data labeling needs. This shows that all data may not be essential to manage concept drift successfully, but more crucially, that

progressive retraining of the model based on a single instance may be more advantageous to managing concept drift. Even though there are no major differences in performance for both balancing methods, the *undersampling* approach provides similar performance metrics to the *oversampling* method with significantly fewer data in the *active learning* case (i.e., 28% more data, on average, for the oversampling case than for the undersampling scenario).

The system calls feature set yielded the worst performance models among all tested models in both evaluated metrics, the number of labeled samples needed and performance achieved (i.e., below 85%). However, when the single instance query strategy is utilized, and more especially, when the uncertainty-based active learning approach is applied, the model is significantly improved, achieving equivalent performance to the baseline model, and even outperforming it when undersampling is employed. Despite that, the labeling needs for the uncertainty-based active learning, which, again, minimizes the labeling cost, is superior than for the permissions case for both balancing techniques (i.e., a minimum of 19% of the data has to be labeled by the oracle). Random selection reaches similar performance as the uncertainty-based strategy but requires, in both cases, over 66.7% of the whole data to be labeled.

The hybrid feature set, which combines the permissions and system calls sets, provided the best overall models, in all cases. The active learning approach using the uncertainty criterion reaches a slightly lower performance than the baseline performance using the batch approach. However, in this case, the benefits of the active learning approach are especially evident for both balancing techniques. The labeling needs are significantly reduced, not over-passing the 9% of the whole data available. As a result, they provide the best performance-labeling trade-off results among all the test scenarios. In this regard, the best model of all the tested scenarios is obtained using the active learning approach combined with undersampling, yielding an average 91.7% F_1 score and 91.6% accuracy using, on average, only 1,460 samples (i.e., $\approx 2.2\%$ of the total data) to provide effective detection in the seven-year-long study period. Comparatively, the uncertainty-based active learning approach for the hybrid-featured models requires 10–15 times fewer data than the random query approach to achieve better performance results and 50 times fewer data to reach similar detection performance than the baseline models. These results show that the hybrid feature set generates better discriminatory models which benefit notably from the active learning approach, being able to handle concept drift with a very reduced quantity of labeled data belonging to specific time frames along a seven-year-long time span (i.e., from September–October 2011 to May–June 2018).

To further explore the results, the summary values reported in Table 16 are provided in a more fine-grained detail over the whole analyzed historical timeline in Fig. 47, 48, 49, 50. In these figures, the X-axis reports the time frame of the specific data chunk, encompassing, at maximum, 2 months of data. The labels provide the year and month separated by a slash (e.g., 2011/9-10 reports data comprised between September and October 2011). The left Y-axis reports the number of samples included in every data chunk (i.e., grey color), thus composing the unlabeled pool of samples for the active learning approaches, that were actually labeled by the oracle (i.e., blue color). The reported values for the number of labeled samples (i.e., blue area on the bars) are mean values with the confidence interval of the mean estimation reported by the white whiskers that extend above and below the mean (i.e., 95% confidence level) due to the degree of randomness of the approaches used. The average performance scores obtained on each data chunk are reported by the yellow (i.e., accuracy) and blue (i.e., F_1 score) lines placed on top of the bar chart, and

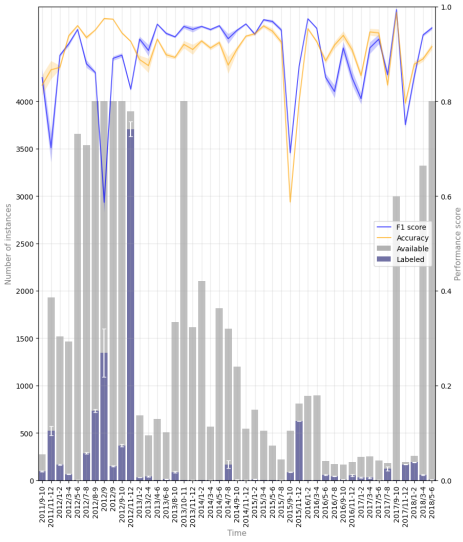


Figure 47: Active learning results for permissions and undersampling [36]

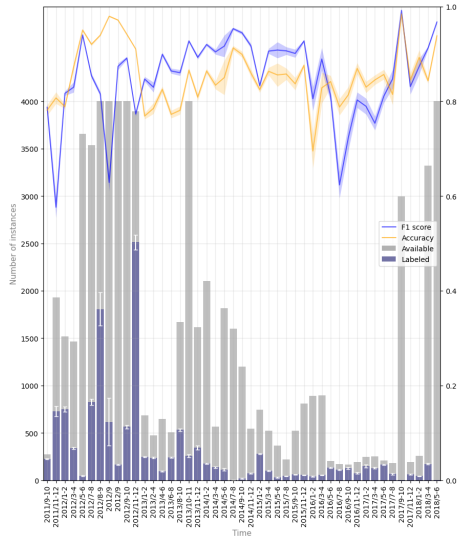


Figure 48: Active learning results for system calls and undersampling [36]

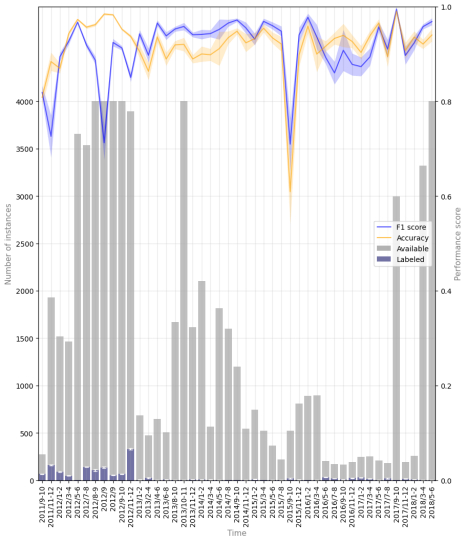


Figure 49: Active learning results for hybrid features and undersampling [36]

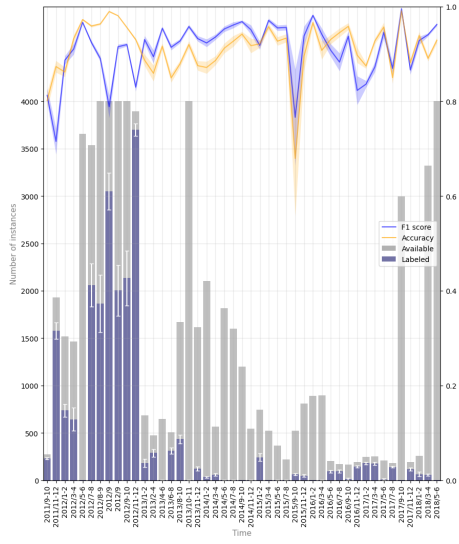


Figure 50: Random selection strategy results for hybrid features and undersampling [36]

ranging from 0 to 1 (i.e., right Y-axis). The standard deviation of these performance metrics is provided by the colored ribbons surrounding the average lines. Fig. 47 provides the average results for the uncertainty-based active learning approach when undersampling and the permissions set were used. Fig. 48 reports the same information when the system calls set was used, while Fig. 49 provides the hybrid feature set-related information. These figures enable us to compare the impact of the feature set under the same conditions (i.e., uncertainty-based active learning approach using undersampling). Lastly, Fig. 50 enables the comparison between the best active learning model (depicted in Fig. 49) and the random query strategy for the same feature set and sampling technique.

As can be observed in Figure 47, the permissions feature set enabled the handling of concept drift using significantly less labeled data than the system calls feature set, depicted in Figure 48. With minor exceptions (e.g., 11-12/2012), the permissions feature set required fewer labeled data per chunk to sustain the training target of 95% F_1 score, over-passing this score in many chunks, thus no data was labeled for training purposes (e.g., 10-11/2013, 11-12/2013, 1-2/2014, 3-4/2014, and 5-6/2014). Despite the effectiveness shown by the permissions feature set to handle concept drift using the active learning approach, these results are significantly outperformed by the hybrid feature set, which combines the system calls and permissions feature set. In this case, a reduced proportion of the chunk data is labeled in every chunk to achieve high-performance metrics (e.g., 9-10/2011, 11-12/2011) with extended periods of no new labeled data needs (e.g., from 4-6/2013 to 7-8/2015). Therefore, the high-dimensional feature space created by combining the two feature sets allowed concept drift to be handled better than with any other approach while maintaining high-performance metrics with few samples labeled per chunk. Even though this feature set reduces the data needs in all approaches and strategies, the uncertainty-based query strategy shows significant improvement with respect to the ran-dom query selection, as can be seen in Fig. 50. The random query strategy requires significantly more labeled data per data chunk to sustain performance and address concept drift, evidencing the superiority of the uncertainty-based selection over random query selection.

The obtained results show that the active learning approach, in its most basic form (i.e., uncertainty sampling) can be effectively used to handle concept drift, keeping high-performance metrics while minimizing the data labeling efforts (i.e., the quantity of labeled data needed to keep high performance). As a result, active learning might be an efficient and effective solution to handle concept drift in environments where a large quantity of unlabeled data is available but with high labeling cost. The active learning strategy allows focusing the labeling effort on the *relevant* data to improve the model and discard the *irrelevant* data samples that may not provide any benefit to the model. The comparative performance metrics obtained demonstrate that the gradual modification of the decision boundary caused by the addition of a single relevant sample to the training data set can yield high-performance models using significantly fewer data samples than the batch retraining approach. Compared to the batch retraining approach, random instance selection enhances the labeling requirements. Still, the active learning approach greatly outperforms both of them. Random selection requires consistently more data to achieve roughly the same (but not better) performance metrics than the uncertainty sampling approach. This fact evidences the goodness of the active learning approach to induce great performance models with significantly fewer data needs.

Finally, regarding the impact of the balancing technique used, it can be argued that even though both approaches worked similarly for random and batch strategies, the undersampling approach provided distinctive benefits using the active learning approach when the permissions and hybrid feature sets were used. This technique minimized the data labeling efforts significantly while producing great discriminatory results.

11.4 Chapter summary

To the best of our knowledge, this was the first study that leveraged *active learning* to handle concept drift in Android malware detection. Our results show that the active learning approach, in its most basic form, enables effective concept drift handling in Android malware detection and significantly reduces the data labeling needs. Consequently, it becomes an interesting option to enhance the ML-based detection systems in cybersecurity environments (e.g., malware protection companies, SOCs dealing with Android malware detection), where a large body of unlabeled data is constantly available but the high label-ing cost associated makes the task infeasible and prohibitive, thus affecting the detection capabilities of the system.

Part II

About IoT Botnet Detection

12 IoT botnet attack detection

This chapter summarizes the contributions of this dissertation regarding IoT botnet attack detection research and contextualizes the relation among Publication XI, Publication XII, Publication XIII, Publication XIV, and Publication XV. More precisely, it describes the IoT botnet life cycle as a time-dependent cyclic process where IoT botnet evolution occurs and how the aforementioned publications relate to it.

12.1 The IoT botnet life cycle

The ubiquity and poor security measures of IoT devices make them an enticing target for cyber attackers. Once vulnerable devices are compromised, they become part of a *botnet*. Large IoT botnets are used to perpetrate massive cyber attacks, from massive SPAM campaigns to DDoS attacks, that can cause massive financial losses for companies by disrupting the availability of targeted servers, services or networks for lengthy periods of time. As a result, the vast majority of IoT botnet-related research focuses on attack detection, a late but a critical phase in the botnet life cycle.

An IoT botnet is just a particular type of botnet where the members of the botnet are internet of things devices, instead of computers as in regular botnets. Any type of botnet shows a similar set of phases during its existence, which are referred to with the term *botnet lifetime cycle* or *life-cycle* [50]. In this regard, the botnet lifetime cycle encompasses four stages: formation, command and control (C&C), attack, and post-attack [66]. They are briefly described as follows [50, 66]:

1. *Formation*: A vulnerable device is compromised and infected by a master, thus becoming a member of a botnet under the control of a botmaster. Also referenced as the spreading or injection phase.

2. *Command & Control (C&C)*: A C&C channel is used by the botmaster to establish communication with the bots. The channel is implemented using different protocols and applications such as HTTP, P2P, or IRC. Commands are sent to instruct the bots about required actions, such as launch attacks.

3. *Attack*: After the reception of an instruction, the perpetration of the attack by the botnet members is performed. The main objective of a botnet is to launch *massive* distributed attacks. This phase is also referenced as the *application phase*.

4. *Post-attack*: After the attack and exposure to the defender, some bots might be cleaned from the infection (e.g., patched vulnerability). Therefore, the recruitment of more members is needed in order to keep or increase the size and capabilities of the botnet. For such a purpose, scanning attacks are performed. The newly recruited bots might be merged with the non-exposed bots and the still operational bots to create a new botnet. This *new* botnet will then receive new instructions to perform attacks via the C&C channel and the cycle will repeat.

The first three steps can be understood as the core components of the botnet lifetime cycle, whereas the last step re-initiates the formation step with the objective of overcoming the eventualities that occurred after the attack phase thus enhancing the botnet's population.

Due to the nefarious and massive consequences of IoT botnet-based attacks, most of the related research focuses on attack detection as it is the first step for attack mitigation. In this regard, Publication XI and Publication XII, address relevant aspects for the generation of more effective attack detection models. Publication XI analyzes the application of

hybrid feature selection models to enhance the detection capabilities of the model while reducing the computational needs and increasing the model's interpretability, whereas Publication XII deals with the model's interpretation as an essential means to increase experts' understanding of the model's output and effective attack detection.

12.2 Hybrid feature selection for enhanced IoT botnet attack detection

Feature selection is an integral and important step in the machine learning workflow [1]. By reducing data dimensionality and selecting the most discriminatory subset of features, the classification performance can be boosted while reducing computational needs, avoiding issues related to the *curse of dimensionality*, and increasing the model's explainability (i.e., reducing the model's complexity). Besides, it enables faster training and helps to reduce *overfitting* issues. *Filter* methods are usually the preferred techniques for performing feature selection. However, most of these methods focus on individual scores of data features without considering the relation between them. In this regard, *wrapper* methods, which involve the usage of a specific subset of features tailored to the machine learning algorithm used to induce the model, can help to enhance the model's performance significantly. However, as the feature selection process is reduced to a search problem of the optimal subset of features, it can be computationally expensive. As a trade-off between both approaches, the combination of filter techniques with wrapper methods (i.e., hybrid methods) may constitute a significant improvement.

Publication XI focuses on the analysis of the impact of filter, wrapper, and hybrid feature selection techniques on the detection accuracy of ML models for IoT botnet attack detection.

The data set used in this research was *N-BaIoT* which contains normal and malicious IoT traffic from 9 IoT devices, gathered simulating distinct attacks using *Mirai* and *BashLite* malware [76]. The data points for each data category were randomly selected and normalized. The data set was balanced, so that each of the three class labels (i.e., normal, Mirai and BashLite) were represented in the same proportion. The data set was split into three folds. Two folds were used in the feature selection phase (i.e., development folds) and one fold as a test set in the final stage.

The impact of each feature selection technique used was analyzed for multi-class detection models using cross-validation and macro-averaged F_1 score performance metric. The final models were tested on *unseen* data (i.e., testing set) and the *accuracy* performance metric was reported. The following feature selection techniques were applied independently and compared:

- *Filter methods*: *Fisher's score* and *Pearson's linear correlation coefficient (ρ)*.
- *Wrapper methods*: *Sequential Forward Feature Selection* (SFFS) and *Sequential Backward Feature Elimination* (SBFE).
- *Hybrid methods*: a two-step feature selection procedure where the output of a filter method is used as input for a wrapper method. Both filter methods were combined with both wrapper methods, resulting in four possible cases.

As wrapper-based feature selection may vary according to the classification algorithm used, two widely used multi-class classification algorithms were evaluated: *k*-Nearest Neighbors (*k*-NN) and Random Forest (RF).

A detailed report of the subset of features selected by each feature selection technique and the F_1 score performance yielded by the related models using cross-validation

in the development folds are provided in Publication XI. The testing fold accuracy results are reported in Table 17. The first two columns specify the filter and wrapper method used (i.e., specified as - if none). In the case of the wrapper and hybrid methods, the selected subset was tested using the two classifier algorithms, not just on the specific algorithm used in the feature selection method step to obtain the corresponding subset (i.e., cross-classifier tests). The source of the wrapper subset is provided after the name of the wrapper method utilized (e.g., SFFS *k*-NN). The k-*NN* and *Random Forest* columns provide the accuracy performance on the testing set of each classification algorithm using the selected subsets by each of the feature selection techniques used.

Table 17: Accuracy comparison of all models in the testing set [36]

Filter method	Wrapper method	k-NN	Random Forest
-	-	0.9536	0.9985
FS	-	0.9968	0.9990
ρ	-	0.9224	0.9852
-	SFFS k-NN	0.9982	0.9608
-	SFFS RF	0.9986	0.9988
-	SBFE k-NN	0.9984	0.9788
-	SBFE RF	0.9974	0.9992
FS	SFFS k-NN	0.9994	0.9992
	SFFS RF	0.9938	0.9994
	SBFE k-NN	0.9990	0.9992
	SBFE RF	0.9992	0.9992
ρ	SFFS k-NN	0.9912	0.8475
	SFFS RF	0.9622	0.9972
	SBFE k-NN	0.9906	0.9958
	SBFE RF	0.9013	0.9970

As can be observed in Table 17, high accuracy values (i.e., over 98.00% in most cases) were obtained using filter methods and wrapper methods alone, supporting the goodness of feature selection methods to achieve high-performance metrics with reduced input data. However, the highest accuracy values were obtained using the combination of filter and wrapper methods, the so-called *hybrid* techniques. More specifically, the use of Fisher's score with any of the wrapper methods and classifiers, provided the best performances in almost all cases, yielding over 99.90% accuracy, even in the cross-classifier tests. In this regard, the best results were achieved using the RF as a classifier except in the cases in which SFFS is utilized in combination with k-NN as a wrapper model.

The hybrid feature selection technique may be seen as a trade-off between the simplicity of the filter methods and the more computationally demanding wrapper techniques. The experimentation demonstrated that hybrid feature selection allows for reducing the computational load of the wrapper techniques without any significant loss in detection rates of the machine learning classifiers.

12.3 Understanding the decision: building trust and enhancing detection

The priority of the ML-based studies in the cyber security domain focus on the optimization of the detection model's accuracy. Generally, the ML model is treated as a *black box* where the hyper-parameters and input data are fine-tuned aiming for the maximum performance possible. While this is the most desirable output and unique goal for many ML applications, some fields may require that the humans understand the rationale behind the decision in order to take appropriate subsequent actions (e.g., health decision, incident investigation). In these latter cases, model explainability is crucial to enhance the trust of the experts in the system and the overall success of the system. Intrusion detection is one of these fields where the enrolment of the expert in the ML workflow is critical

for the investigation of relevant incidents and overall success of the human-machine detection system.

Although feature selection methods can make the models more interpretable, it does not guarantee their acceptability in all cases (i.e., experts may not trust over-simplistic models in some situations). In this regard, *local* interpretation methods are specially designed to provide the reasoning behind individual predictions in compelling ways, which may boost the confidence of the expert in the system output.

In general, the application of the classical machine learning techniques presumes that feature selection was conducted. Model-agnostic and post-hoc local interpretation methods are applied to the outputs of the learning models. In this regard, feature selection is a prior step and local interpretation is a posterior step after model generation. As a result, even though they occur in different stages, feature selection and local interpretation may interplay in the whole machine learning workflow, which can have an impact on the quality of the interpretation.

Publication XII analyzed the impact of feature selection on the detection accuracy and the quality of the interpretation. In the first stage, the impact of hyper-parameters and feature selection on data accuracy was explored. In the second stage, the impact of feature selection on the interpretation results was evaluated. Here, we introduced a quality metric for the interpretation results from the cyber security analyst perspective, based on the entropy notion, which assumes that an ideal explanation should be used as an explanation for only one category, as more than one could create confusion for the analyst. Model decisions' explanations were obtained using the Local Interpretable Model-agnostic Explanation (LIME) method [84].

As in Publication XI, the data set used for this research was *N-BaIoT*, composed of labeled IoT botnet attacks and normal network traffic data. The attack data encompasses attack and post-exploitation activities of the botnet life-cycle. The features were ordered according to Fisher's score value and used to find the optimal subset and model hyper-parameters for *k*-Nearest Neighbors, Decision Tree, and Random Forest algorithms. The optimal performance for all classification algorithms was achieved using the top 10-15 features. A more detailed explanation of the feature selection process, the hyper-parameter optimization performed, and their results are provided in Publication XII. The following paragraphs focus on the interplay between the feature selection and interpretation quality.

The application of LIME to 50 randomly selected instances for each class showed that when a small subset of features was used (e.g., 3 features), the same explanations, expressed in the form of inequalities as shown in Table 18 (i.e., x_1, x_2, x_3 refer to three input features), were used to explain different categories. This fact was observed for all the classifiers used.

Table 18: Class distribution for each explanation - k-NN model [40]

Explanation Rule	Benign	Bashlite	Mirai
$x_1, x_2 \leq 277.96$ and $x_3 \leq 112.94$	50	7	7
$x_1, x_2 > 679.91$ and $193.25 < x_3 \leq 246.09$	0	4	0
$x_1, x_2 > 679.91$ and $x_3 > 246.09$	0	26	7
$x_1, x_2 \leq 277.96$ and $x_3 \leq 193.25$	0	1	0
$277.96 < x_1, x_2 \leq 595.68$ and $112.94 \leq x_3 \leq 193.25$	0	1	12
$277.96 < x_1, x_2 \leq 595.68$ and $193.25 < x_3 \leq 246.09$	0	0	4
$595.68 < x_1, x_2 \leq 679.91$ and $193.25 < x_3 \leq 246.09$	0	0	12
$595.68 < x_1, x_2 \leq 679.91$ and $112.94 < x_3 \leq 193.25$	0	0	3
$x_1, x_2 \leq 277.96$ and $112.94 < x_3 \leq 193.25$	0	0	1
$277.96 < x_1, x_2 \leq 595.68$ and $x_3 \leq 112.94$	0	0	1

As can be observed in Table 18, certain explanations overlap between categories, that is, the same explanation is provided to explain different class predictions. For instance, the same rule (i.e., $x_1, x_2 \leq 277.96$ and $x_3 \leq 112.94$) explains all benign predicted samples, but also explains 7 decisions for each of the malware categories. Thus, in this case, the same inequality rule may explain the three categories. On the other side, the second rule, $x_1, x_2 > 679.91$ and $193.25 < x_3 \leq 246.09$ explains 4 Bashlite instances and no other category, thus no overlapping occured. The usage of the same explanations to explain different class predictions would create confusion among security analysts even with a highly accurate model. In this regard, hyper-parameter selection, including the number of selected features, applied at each step before the post-hoc interpretation has an impact on the overlaps of the local explanations. In this study, the implications of these choices were analyzed rather than the decision quality of the interpretation method itself.

We introduced an interpretation quality metric, provided in Eq. 5, which computes the degree of explanation overlap by using the entropy notion. Let e_i be the i-th explanation in an explanation set E, where K is the number of categories and p_k is the ratio of instances labeled by category k to all the instances described by e_i.

$$e_i = \sum_{k=1}^{K} -p_k \cdot \log_2 p_k \tag{5}$$

The value of $p_k \cdot \log_2 p_k$ is considered as zero when p_k is zero. Eq. 5 gets the lowest value (i.e., zero) when all instances explained by one inequality rule belong to the same category (i.e., explanation overlap is zero) and provides the highest value when the explained instances are equally distributed among the categories. Therefore, the first, third, and fifth rows in Table 18 show entropy values distinct to zero, whereas all others, are exactly zero.

Let's assume that we have N instances and we apply LIME to provide an explanation for each instance. The explanation overlap, EO, of an entire explanation set having N elements is computed as follows:

$$EO = \sum_{i=1}^{N} e_i \tag{6}$$

where e_i is computed as in Eq. 5.

Figure 51 shows the explanation overlap (EO) of a randomly selected balanced instance set, explained using LIME for decisions taken by learning models created with DT, k-NN and RF algorithms and a varying number of features. The X-axis of the graph gives the number of features used to create the model, ordered according to the Fisher's score, and the Y-axis demonstrates the value of explanation overlap obtained according to the chosen number of features (i.e., using Equation 6). Figure 52 provides the accuracy performance (i.e., y-axis) based on the number of ranked features used.

The results show that all ML algorithms reached the zero value for EO between 13–17 features, meaning that, at the *post-hoc* interpretation step, LIME requires at least such number of features to assign one explanation to just only one category. If the machine learning model utilizes fewer features, the explanations may confuse the analysts as one inequality set may explain more than one category. However, even though the models reached the zero overlap value in the 13–17 features range some fluctuations exist for the models after this range, especially for $k-$NN and RF algorithms. Despite that, as the models reached optimal accuracy using 10–15 features, in our case, it is possible to have a clear explanation rule set and optimal accuracy with 13–17 features. However, such a number of features might not be comprehensible by the experts as the feature set may have too

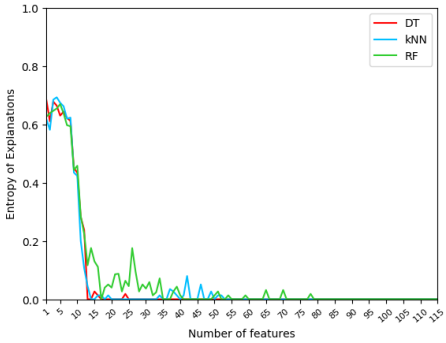


Figure 51: EO using Fisher's score ranking [40]

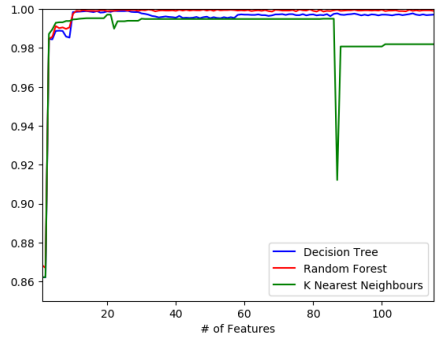


Figure 52: Accuracy using ranked features [40]

many inequalities. Miller's psychological theory states that humans can handle 7 ± 2 abstract entities at the same time [77]. Therefore, 13–17 features may not be suitable for expert understanding despite the high detection accuracy rates. Additionally, the similarity of the data features in the data set used, and reflected in the Fisher's score ranking, may generate additional comprehensibility issues. In this regard, *wrapper*, *hybrid* or *embedded* feature selection methods may yield better accuracy rates with less number of features, as demonstrated in Publication XI. Even though the comprehensive analysis of feature selection methods was out of the scope of the paper, an additional experiment was conducted to see the results using a *customized* feature set which we selected as follows. We traversed the list ranked by Fisher's score value and selected 20 features that belonged to different feature categories. This selection procedure meant that the final list was more varied but that included features with lower Fisher's score value.

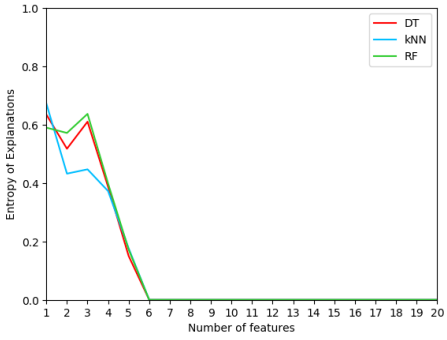


Figure 53: EO using the custom feature sets [40]

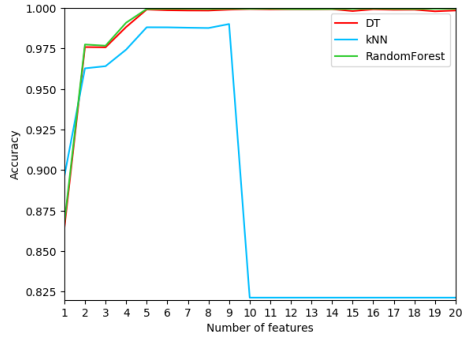


Figure 54: Accuracy using the custom feature sets [40]

The explanation overlap and accuracy results for the *custom* feature set are given in Figure 53 and Figure 54. It is worth highlighting that the *EO* of all models reached zero value with only six features, and no fluctuations were observed after it. The accuracy values shown in Figure 54 demonstrate that optimal detection accuracy was reached with 5 features, showing similar performance as when Fisher's score ranking was used. However, in this case, the number of selected features is within the range of human capabilities stated in Miller's theory. Besides, as the features in the *custom* feature set belong to different categories, it can be argued that the security analysts could perceive better the

interpretation rules and grasp the distinctions between benign and malware data more easily.

Although we did not investigate more feature selection techniques, the size reduction of the optimal feature set from the accuracy point of view could be attributed to possible dependencies among features or it could be argued that a filter method, which is computationally cheap, was not enough. In any case, the quality metric proposed in this study, i.e., explanation overlap, supported the interpretability analysis of the selected features.

12.4 Chapter summary

This chapter presented the usage of feature selection methods to induce better models for IoT-based attack detection and the usage of post-hoc interpretation techniques to understand the rationale behind the classifier decisions as a means to build trust and enhance detection. Besides, an entropy-based metric was proposed to evaluate the explanation overlap and enable the assessment of the impact of the feature selection technique used on the *quality* of the explanations obtained in the post-hoc interpretation phase and the detection performance.

13 IoT botnet attack prevention

The mitigation of IoT botnet attacks, which may create havoc and severe financial losses for companies and individuals, is a priority and the focus of the related research. However, little to no attention has been brought to attack prevention, that is, early detection of IoT botnet formation. The vast majority of the research in the problem domain and the available data sets are centered around the attack and post-attack stages; however, to prevent attacks, effective detection should be performed in the early stages of the botnet life cycle (i.e., infection and C&C stages, as described in Section 12.1).

Publication XIII, Publication XIV, and Publication XV revolve around the concept of IoT botnet attack prevention by focusing attention on the early stages of botnet deployment. More specifically, as most of the data sets available are centered around the attack phase, Publication XIII and Publication XIV introduce and describe the generation of *MedBIoT*, a novel IoT data set focused on the early stages of the IoT botnet life cycle, and its usage to induce ML-based detection models that can effectively detect and discriminate between normal and malicious IoT network traffic. Publication XV combines *MedBIoT* and the *active learning* approach to improve the learning process and generate more efficient and effective detection systems by including a human *oracle* in the process.

13.1 MedBIoT: early stage IoT botnet data set

The available data sets for IoT botnet detection show similar characteristics. Namely, they are collected in small-sized IoT networks, focus on attack simulation, and use a small variety of either real or emulated IoT devices. This makes the induced models very specific and precise to detect attacks but not to prevent them or detect botnet formation. As machine learning models rely on data quality and quantity, using only attack data limits what the models can learn and detect. Therefore, if early detection is the objective, a distinct data type must be used for such a purpose. *MedBIoT* data set addresses these issues and the existing research gap by providing a data set that enables the generation of learning models for enhanced intrusion detection systems (IDS) capable of detecting IoT botnets at early stages and, consequently, preventing IoT botnet attacks.

The experimental setup and generation of malicious behavioral data sets were carried out in [75] as the main contribution and work of the thesis. Detailed information about the network topology, experimental setup and emulated behaviors is provided in [75], Publication XIII and Publication XIV. Based on the generated behaviors, Publication XIII introduces *MedBIoT*, a novel IoT botnet data set that includes malicious and normal network traffic data and addresses the research gaps of the existing IoT data sets. More specifically, *MedBIoT* was acquired on a medium-sized IoT network architecture (i.e., 83 devices), where real and emulated IoT devices were deployed and infected with three prominent IoT botnets (i.e., Mirai, BashLite, and Torii). The extended size of the IoT network used to collect *MedBIoT* data set enabled the capture of malware spreading patterns and interactions that cannot be observed in small-sized networks, providing a more realistic environment. Furthermore, none of the previous data sets used the combination of emulated and real devices in the same network. Additionally, this data set includes the behavior of Torii botnet malware, being the first publicly available data set to deploy it. Lastly, as this data set focuses on malware infection, propagation and communication with C&C server phases, the first stages of botnet deployment, it can be seen as a complement of the already available data sets, focused on attack data, to build an integral and enhanced IDS for IoT networks. In this regard, the data set is provided in structured and raw format. The structured format provides extracted features from the network traffic

captured, in tabular data format, like the ones generated in other data sets (i.e., *N-BaIoT* and *Bot-IoT* features), which enables the comparison and immediate usage of the data sets together in an integral approach to the IoT botnet life cycle. The raw format provides the captured network packets (i.e., *pcap* files) for further manipulation and generation of custom data features by the users. Besides, the data set is provided in *bulk* format (i.e., all packets of the same class label are included together in the same file) and in *fine-grained* format (i.e., the data are provided in separated files for each data source, life-cycle phase, and device type).

Publication XIV is an extension of Publication XIII. It provides further experimentation on the usage of *MedBIoT* data set to build effective ML-based IDS.

A brief summary of the data set features and the machine learning-based experimental results, further detailed in Publication XIII and Publication XIV, are presented in the following paragraphs.

13.1.1 Data set features

The distribution of the network data that compose *MedBIoT* is reported in Table 19. The total size of the data set is 17,845,567 network packets. The majority of the traffic that compose the data set corresponds to benign or normal IoT traffic (i.e., 70.27%), whereas the majority of the malware traffic corresponds to BashLite.

Table 19: Data set composition [48]

Data category	Devices	Nr. of packets	Proportion
Normal	83	12,540,478	70.27%
BashLite	40	4,143,276	23.22%
Mirai	25	842,674	4.72%
Torii	12	319,139	1.79%
All	All	17,845,567	100%

A detailed analysis of the captured data showed that 32% of the normal network traffic was related to system updates, 53% to device communication (i.e., MQTT protocol), and 15% to other network data (e.g., TLS errors, ping, etc). Regarding the malicious data, 68% of the traffic was related to malware propagation actions, whereas the remaining 32% to direct communication between the bots and the C&C servers that were deployed in the experimental setup.

13.1.2 Early IoT malicious behavior detection

In order to test the goodness of the data set to induce effective intrusion detection systems for early IoT botnet detection, four different scenarios were tested. Two scenarios involved typical *supervised learning* approaches and two used *unsupervised learning* models. The experimentation performed is explained as follows.

- ***Supervised learning***: *binary* and *multi-class* classification models were induced using randomly selected data points from the source data set to generate a reduced balanced data set. *k*-Nearest Neighbors, Decision Tree, Support Vector Machines, and Random Forest algorithms were used as classifiers. Cross-validation was used to assess the performance of the models. Relevant performance metrics were retrieved.

  - *Binary classification*: the classification task involved the discrimination between normal and malware data. The malware category was composed of

an equal number of instances from the three malware categories. The results of the binary classification models are reported in Table 20.

Table 20: Binary classification models' performance [44]

Model	Accuracy	Precision	Recall	F_1 score
k-NN	0.8871	0.9034	0.8871	0.8842
DT	0.9541	0.9582	0.9541	0.9538
RF	0.9702	0.9731	0.9702	0.9700

As shown in Table 20, the RF model provided the best discrimination accuracy, classifying correctly over 97% of the network traffic. k-NN and decision tree models showed lower discriminatory capabilities. Linear SVM results are not reported as they provided a poor performance on all metrics. This fact may suggest that the data is not linearly separable; thus, linear classifiers such as SVM with linear *kernel* or Logistic Regression may not be suitable for the binary classification task using this data set. Nevertheless, the results obtained using the other algorithms evidence the effective capabilities of ML approaches to detect botnet malware traffic in the initial stages (i.e., infection, propagation and communication with the C&C server stages) and disregarding the malware type. Furthermore, it was demonstrated that *MedBIoT* is suitable to be used as a medium-sized realistic IoT data set for IoT botnet detection scenarios, and IDS training and testing purposes.

– *Multi-class classification*: this classification task involved the recognition of the four *class* types; thus, the data set was divided into four classes or labels according to the data source: normal, Mirai, BashLite, and Torii. Four-class or multi-class classification models were induced, and 10-fold cross-validated using the same algorithms as in the binary task. An equal number of data samples were used for all categories. The purpose of this task was not only to test the classification capabilities of legitimate and malware classes but also the recognition of the specific malware source. Table 21 shows the results obtained for this task. Like in the binary approach, SVM algorithm is not reported, as it showed a poor performance in all metrics.

Table 21: Multi-class classification models' performance [44]

Model	Accuracy	Precision	Recall	F_1 score
k-NN	0.8990	0.9073	0.8990	0.8958
DT	0.9379	0.9478	0.9379	0.9347
RF	0.9617	0.9692	0.9617	0.9602

As can be observed in Table 21, in a similar fashion as in the binary models, the Random Forest model outperformed Decision Tree and k-NN algorithms in the multi-class classification setting. RF algorithm provides similar performance for the multi-class task as in the binary task, reporting over 96% accuracy in all metrics. The analysis of the RF model's confusion matrix evidenced that the classification *error* was not significantly biased towards any of the classes. These results suggest that network traffic sources can be effectively discriminated, even in the earliest stages of botnet infection. It also demonstrates that the learning capabilities of ML-based detection methods can be accurate both in the binary detection task and in the detection of different sources of malicious traffic in medium-sized IoT networks.

- ***Unsupervised learning***: this task involved the identification of abnormal or unusual observations (i.e., *anomalies*) in the input data distribution. For this task, malware activity was considered *anomalous* as it does not represent the *normal* behavior of the IoT devices. Two *novelty detection* tasks were generated for this learning type. For this task, it is assumed that the training data set does not contain outliers (i.e., contains only *normal*) and the goal is, given a new observation, to detect whether it can be categorized as an *outlier/novelty* (i.e., malware) or an *inlier* (i.e., normal). The anomaly detection algorithm used was Local Outlier Factor (LOF), which is capable to perform both novelty and outlier detection tasks. The models were trained only with normal data and tested against normal and malicious data. The data set was standardized and the data dimensionality was reduced using *Principal Component Analysis* (i.e., 10–30 Principal components) prior to model induction.

  - *Novelty detection - first scenario*: in this scenario, the normal data captured during the time frame where a specific malware was running was used to build the models. The testing sets correspond to hold-out normal data and malware data from that specific collection time frame. For example, in the first row of Table 22, the training data corresponds to normal data captured during the deployment of BashLite malware. The testing samples correspond to normal data from the same period of time and BashLite malware-generated data. The detection performances for this first scenario are provided in Table 22. The column *training*, specifies the source of the normal data used to build the corresponding model, whereas the *test malware* and *test normal* refer to the source of data used for testing purposes. The *mixed total* column provides the average of the previous two columns. The *all* value refers to a stratified mix of the normal data. The performance metric reported is accuracy.

    Table 22: Novelty detection - first scenario accuracy performance [44]

Training	Test Normal	Test malware	Mixed Total
BashLite	0.9486	0.9628	0.9557
Mirai	0.9331	0.8552	0.8942
Torii	0.9433	0.9515	0.9474
All	0.9444	0.9129	0.9286

    As can be observed in Table 22, the models built with BashLite, Torii and combined legitimate data provide detection performances of over 91% on malware and over 93% on normal data. Normal data belonging to Mirai deployment provides less accuracy on the malware data and normal test data, suggesting that this malware traffic is more similar to legitimate traffic but prone to be discriminated effectively. According to the results, BashLite malware provides a differentiated profile from normal traffic that makes the models more effective in the detection of this specific malware. Torii and the mixed model (i.e., using stratified randomly sampled legitimate data from the three data sets) provide high accuracy ratios for malware detection. In any case, these results evidence that IoT malware traffic can be discriminated from legitimate traffic and effectively detected using anomaly-based detection models in the early stages of a botnet deployment, prior to the attack phase.

  - *Novelty detection - second scenario*: for this second scenario, the same models built on the first scenario were tested against other testing sets belonging to different malware data. For example, in the first row of Table 23, the training

data corresponds to legitimate data acquired during the deployment of BashLite malware. The testing set corresponds to data acquired in the same time frame, thus BashLite-generated data, but also other testing sets are used such as malicious data belonging to the deployments of Torii and Mirai malware. This setting allows testing the goodness of the anomaly detection models to detect different types of malware. The performance results are provided in Table 23. In this table, the column *training* specifies the source of the normal data used to build the corresponding model, whereas the rest of the columns specify the source of the malware data that was tested. The *test mixed* column provides the performance of the model on a mixed and balanced data set containing the three malware data. The performance metric reported is accuracy.

Table 23: Novelty detection - second scenario accuracy performance [44]

Training	Test Mirai	Test Torii	Test BashLite	Test Mixed
BashLite	0.9066	0.9842	0.9628	0.9536
Mirai	0.8552	0.9665	0.9643	0.9262
Torii	0.8839	0.9515	0.9618	0.9120
All	0.8407	0.9594	0.9615	0.9074

The results provided in Table 23 suggest that the anomaly-based detection models built in the first scenario are capable to detect effectively not only its related malware but also the other sources of malware. With the exception of the detection of Mirai malware, which is slightly worse than for the other malware, the detection ratios are over 91% in all models, disregarding the data source used. These results emphasize the goodness of the anomaly-based models to detect malware effectively and the goodness of the generated data set to build effective anomaly-based IoT malware detection models in the early stages of botnet deployment.

13.2 Active learning for early IoT botnet detection

Active learning is well-suited to problem domains in which collecting data is easy and cheap, but data labeling is expensive. Intrusion detection could be considered one of these domains as it is easy to collect raw network or host data from the systems and convert it to a suitable format for learning tasks. However, assigning the relevant resources to the labeling tasks is difficult due to the limited number of human experts with sufficient security skills to perform the task. Besides, confidentiality concerns usually prevent organizations to share any data, be it raw or labeled, with others, exacerbating the problem.

The active learning approach was already introduced in Section 11.1. In the study described in Chapter 11 the active learning approach was leveraged to deal efficiently with concept drift in Android malware detection. Intrusion detection is another problem domain that can benefit from active learning.

The work explained in this chapter is based on Publication XV, which is currently undergoing a peer-review process. This work has been included in the thesis to support the related research and make a more complete narrative of the whole research performed.

Publication XV explores the application of active learning to the intrusion detection domain by simulating a realistic scenario in a SOC where human experts are available to act as oracles and the wealth of data processed makes it unfeasible to allocate the needed resources to label all the incoming data. More precisely, the predictive performance of the active learning approach is evaluated by assessing the impact of the quality of the

instance selection process (i.e., query strategy), the size of the unlabeled data pool, the number of labeled instances used to create the initial model, and the accuracy of the expert decisions.

The usage of the *MedBIoT* data set enabled us to test the application of the active learning approach to early IoT botnet detection which might be of special interest for SOC environments aiming for attack prevention. Redundant features were removed and binary classification models were induced. After the generation of the baseline classifier model using different subsets of data (i.e., passive approach), used as a reference for the active learning approach, different scenarios were designed to evaluate the impact of distinct variables on the success of the active learning implementation. The baseline model, the description of the scenarios and the main results are described in the following paragraphs.

13.2.1 Baseline model: the *passive* approach

The baseline classification algorithm, Random Forest, which outperformed other algorithms in initial tests, was used to build classification models using training sets of different sizes. For this purpose, training subsets of different sizes were sampled randomly from the whole training set, without replacement. For every training set size used, 50 models were induced and the results were averaged using mean and standard deviation. The testing set was the same in all cases. The results are provided in Figure 55. In this graph, the horizontal axis provides the training data set size and the vertical axis the accuracy performance obtained on the testing set. The blue line provides the average accuracy value, while the standard deviation is reported by the surrounding ribbons.

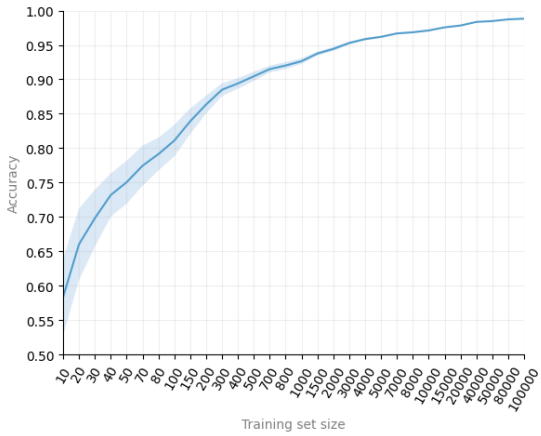


Figure 55: Performance of the baseline model using different training subsets [45]

As shown in Figure 55, in general, the larger the training set size, the greater the accuracy retrieved and the lower the variability observed among iterations. Furthermore, the performance line is steepest for the smaller data sets, suggesting that the model is capable to learn effectively from small training sets and improve its knowledge rapidly with additional data samples. For instance, from 10 to 40 data samples, the performance increases from 0.58 to 0.73, whereas from 10,000 to 40,000 the increase is barely 0.01 points, from 0.97 to 0.98. The greatest performance obtained is 0.988 with 100,000 samples, that is, using the whole training set. The 0.90 accuracy performance is achieved with 500 samples, whereas 0.95 and 0.97 accuracy values are achieved using data sets

containing 3,000 and 10,000 samples, respectively.

The baseline models are *static* models, that is, they are trained using a large amount of data and they are not re-trained after their generation. The baseline models were used as a reference to assess the performance and benefits of the active learning approach.

13.2.2 Active learning scenarios

The most common active learning approach is the pool-based framework, introduced in Section 11.1. We performed a benchmarking of pool-based strategies for active learning. The pool-based active learning framework assumes the existence of a small set of labeled data (L) and the availability of a large collection (pool) of unlabeled data (U) [88]. Instances are selected by the classifier from the unlabeled pool of instances for expert annotation. The labeled sample is then incorporated into the labeled training set which is used to update the knowledge of the learning model (i.e., model retraining). The instances are usually selected greedily based on an *informativeness score* used to evaluate all the instances of the unlabeled pool [89]. Three query strategies were tested in the active learning scenarios. *Uncertainty sampling*, *query by committee*, and *ranked batch-mode sampling* approaches were evaluated using different parameters and informativeness criteria. The following paragraphs summarize the scenarios and provide the main results. A detailed description and further explanation of the query strategies and the informativeness criteria used are provided in Publication XV. All the models evaluated in all the scenarios, including active learning and *passive* approaches (i.e., static models), were tested on the same testing set. Besides, as all the scenarios involved some degree of randomization, 50 iterations were performed per model in each testing scenario. Therefore, the reported performance was the average accuracy score of all iterations. The following items summarize the scenarios and their main results.

- ***Uncertainty sampling***: a single classifier was used to generate an initial detection model that used the active learning approach to update its knowledge based on different query strategies. The query strategies tested were *classification uncertainty*, *classification margin*, and *classification entropy*. A total of 1,000 queries were performed and the accuracy performance of the models was retrieved. In addition, the impact of two variables on the active learner performance was evaluated: the *size of the initial data set* (i.e., seed size), and the *size of the unlabeled pool of instances*. In all cases, random selection was used to generate the initial training set (i.e., seed) and the unlabeled pool samples.

  The results for the models using *classification uncertainty* as query strategy are provided in Figure 56. As can be seen, for any seed size, the maximum accuracy performance is reached with an unlabeled pool of about 7,000 instances (i.e., the red line). The graphs evidence that when the unlabeled pool is too small (i.e., the lightest green line) or too big (i.e., the darkest blue line), the learning is hindered. The rationale behind these observations might be found in the lack of representativeness of the samples in the former case, whereas, in the latter case, an excess of similar data samples (i.e., with the same informativeness value) might cause suboptimal instance selection and, consequently, slow down the learning.

  Regarding the seed size, it is worth noticing that all models' performance is over 0.97, thus disregarding the initial training data set used, the active learners achieved the same high-performance score. The only substantial difference observed among the models is the shape of the curve leading from the initial model to the last query. In this regard, with the exception of the model with an initial seed of 2 instances (i.e.,

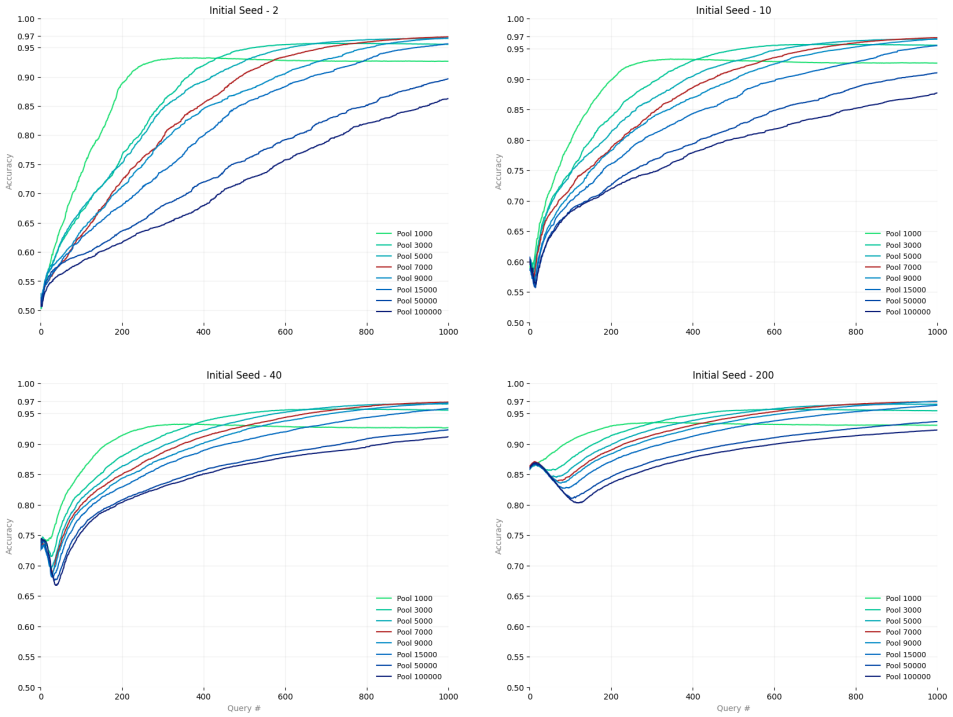


Figure 56: Uncertainty sampling: classification uncertainty score results [45]

top-left graph), the models suffer from an initial decrease in performance, that is corrected later with a boost in performance. This correction takes more time (i.e., queries) for the models induced with more data, that is, a larger initial seed size. This initial decrease may have been brought on by the bias that the initial classifiers introduced, selecting sub-optimal samples for the first queries that possibly responded to the bias in the model but were less capable of *generalization*. In any case, all the active learning models recover from the initial draw-down and surpass the 0.95 performance in about 600 queries and achieve 0.97 accuracy around 800 queries. Comparatively, the baseline model, depicted in Figure 55), achieved 0.95 and 0.97 accuracy scores when 3,000 and 10,000 samples were used, respectively. This shows that the active learner can achieve similar performance to the passive model using 10 to 12 times less labeled data. Besides, the active learning approach seems to provide additional benefits when the initial seed is small, as no initial draw-down in performance is observed in that case. However, the initial accuracy is lower when the initial seed is smaller.

The three query strategies for uncertainty sampling evaluated produced similar results. A comparison of the three query strategies with the pool size of 7,000 and the four initial seed sizes is provided in Figure 57. In this figure, the performance of the random selection approach, in which the query sample is selected randomly from the unlabeled pool, is provided as a comparative baseline to evaluate the effectiveness of the different query strategies. Based on Figure 57, the random approach seems to provide better performance than the active learning approaches in the initial stages. However, as the number of queries increases, the active learner is ca-

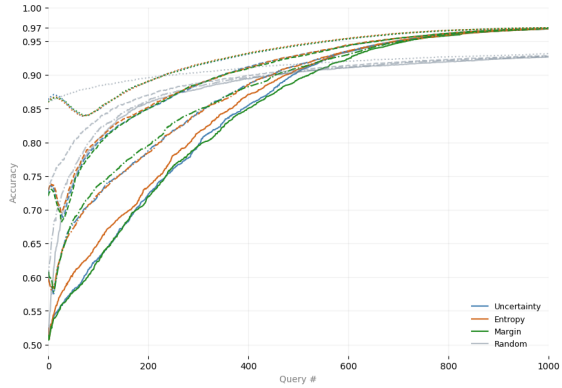


Figure 57: Comparison among the uncertainty sampling strategies and random query selection [45]

pable of learning better and faster using the uncertainty query strategies than using the random query selection. All the active learning approaches, for any seed size, outperform the random query selection strategy after ≈ 500 queries. Besides, the maximum performance of the random approach is lower and plateaus faster than any of the active learning query strategies. More precisely, the best model using the random approach achieved a maximum of 0.93 accuracy score after 1,000 queries, whereas this performance was reached by most of the active learner models in 500 queries, reaching over 0.97 accuracy score after 1,000 queries.

- ***Ranked batch-mode sampling***: the same scenario as for the uncertainty sampling was implemented but the query strategy used was ranked batch-mode. This strategy enables the learner to request for labeling more than one instance per query. Besides, this strategy enhances the informativeness score using an additional scoring metric based on similarity measures to improve query selection. In the testing scenarios, the impact of the *batch size* (i.e., number of instances queried at once), the *pool size* and the *size of the initial seed* were evaluated. Similar to the uncertainty scenario, a total of 1,000 instances were queried in a variable number of queries per model, depending on the batch size. The models' accuracy was retrieved.

  The results obtained using this approach did not show any additional improvement to the uncertainty strategy. More precisely, the ranked batch-mode strategy did not perform better than the uncertainty sampling approach. As a reference, Figure 58 provides the performance results for the batch size of 2 instances (i.e., 500 queries), and different combinations of seed and pool sizes.

  However, an interesting result from the ranked batch-mode testing scenarios is that the best performance was achieved with a smaller pool size than for the uncertainty sampling approach, reaching the maximum performance of 0.95 accuracy with an unlabeled pool of 3,000 instances. As can be seen in Figure 58, the pool of 7,000 unlabeled instances provided significantly worse results. Even though the ranked batch-mode models did not provide improved performance concerning the single query mode (i.e., uncertainty mode), none of the models in Figure 58 show the initial decrease in performance observed for the uncertainty sampling strategies.

  These results suggest that a hybrid approach, using uncertainty-based active learners combined with initial random selection or ranked-batch mode (i.e., just for the

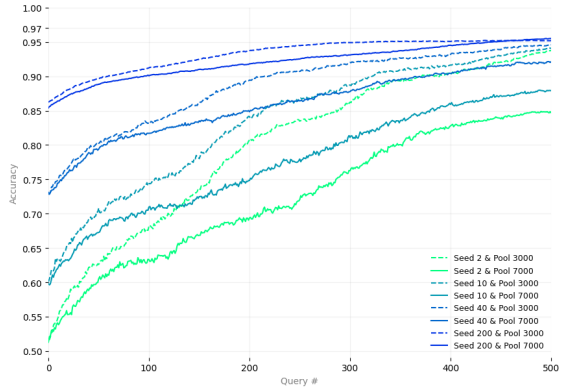


Figure 58: Ranked batch-mode sampling performance for the batch size of 2 [45]

initial queries), may help to avoid the initial dips in performance and overcome the initial bias evidenced by the uncertainty sampling models.

- ***Query by committee***: a group of classifiers was generated and the *query by committee* approach was used to update their knowledge based on different query strategies. The query strategies tested were *vote entropy*, *consensus entropy*, and *maximum disagreement*. The *best* pool size for the uncertainty sampling scenarios was used for simplicity and similarity. The impact of the *initial seed size* and the *size of the committee* (i.e., number of members) were evaluated. As in the uncertainty sampling scenarios, 1,000 queries were performed, and the accuracy was retrieved.

  The results for the *consensus entropy* query strategy are provided in Figure 59. Similar to Figure 56, each graph in Figure 59 shows the performance of the models built with different initial seeds. For each model, distinct committee sizes were tested, indicated as CE in the graph legend. For the sake of comparison, the best model for *maximum disagreement* (MD) and *vote entropy* (VE) query strategies are provided with red color and different line styles. MD and VE query strategies provided significantly lower performance than the CE strategy.

  As can be observed in Figure 59, the larger the committee size, the better the results. The largest committee shows the steepest learning curve in all cases (i.e., faster learning). Besides, a large committee size tends to avoid the decrease in performance on the initial queries. It provides improved learning from the early stages of the active learning process. However, a larger committee implies the retraining of more models after the labeling process, which might be more time-consuming, and demand more resources. In any case, using the CE query strategy and any initial seed size, 0.95 accuracy is achieved before 600 queries and 0.97 before 800 queries. After 800–900 queries, all models seem to converge providing the same performance after 1,000 queries.

  The MD and VE query strategies showed sub-optimal performance, especially in the case of the MD query strategy where the best model, using a committee of 10 members, did not even achieve 0.95 accuracy performance after 1,000 queries. The best VE model, built with 10 committee members, reached 0.97 performance after 1,000 queries but its learning curve is worse than the CE strategy with a committee of 5 members, for any initial seed sizes. Besides, in all cases, the VE strategy suffered

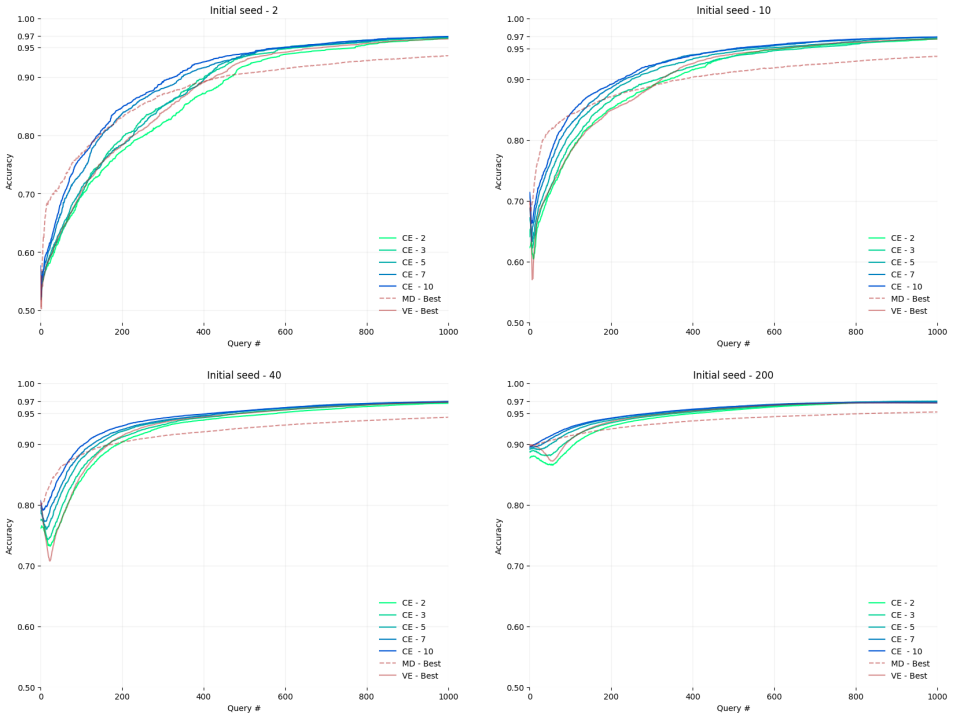


Figure 59: Query by committee performance results [45]

from a pronounced decrease in performance for the initial queries.

Given the similarities between the uncertainty sampling approach and the query by committee (QBC) approach, a direct comparison is well-motivated. The comparison is provided in Figure 60 for different initial seeds. The best uncertainty model and two QBC models (i.e., with 3 and 10 members) are compared in the graph.

As can be observed in Figure 60, both active learning approaches converge to 0.97 performance after 1,000 queries, which shows the goodness of either of the active learning approaches to achieve high performance with a small fraction of the data needed by the passive learning approach to achieve the same results (see Figure 55). However, the learning curves are notably different for both approaches, especially for small initial seed sizes (i.e., greater separation between the curves). The QBC strategy with a committee of 10 classifiers provides the steepest learning curve, achieving high-performance metrics faster than the other approaches. Besides, the initial models for the QBC approach start at a higher accuracy score than the uncertainty sampling models. The *ensemble* of classifiers that form the committee provides improved performance from the initial step, emphasizing the goodness of combining several classifiers for enhanced learning. However, as mentioned before, implementing committee-based approaches requires more resources and may imply longer retraining schedules, especially when the committee size is large. For this reason, the QBC with a committee size of 3 members is a good trade-off between the best QBC model, composed of 10 members, and the best uncertainty model.

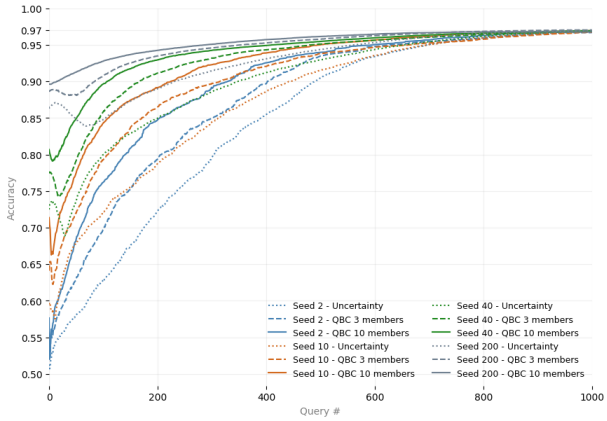


Figure 60: Query by committee vs. uncertainty sampling [45]

13.2.3 Wrong labeling impact

In all the previous scenarios it was assumed that the *oracle* labeling the queries provided a 100% accuracy on class identification, that is, a 0% error in data class imputation. Although highly desirable, this scenario is relatively unrealistic. For example, in SOC environments, a diversity of analysts with different degrees of experience and expertise may coexist. Therefore, a high degree of accuracy can be expected, but a non-null probability of wrong labeling cannot be ruled out. Consequently, the last kind of simulated scenario considered the possibility of mistakes or wrong labeling in the active learning implementation.

To implement the wrong labeling scenarios, the best models from the previous three scenarios were selected and distinct wrong labeling probability values were evaluated (i.e., 5%, 15%, and 25%). This implied that based on the outcome of a random number generator function, a specific mislabeling probability was applied to each query instance. Furthermore, the baseline models, built using the *passive learning* approach, were also induced applying the wrong labeling approach to the training set.

Figure 61 displays the outcomes for the best QBC, best ranked batch-mode, and best uncertainty sampling models when various initial seed sizes and wrong labeling probabilities are applied. As can be observed, the active learners with the smallest initial seeds are the ones affected the most by the wrong labels. This is expected as for the active learners, the wrong labeling is applied to the instance selected by the classifier as optimal for learning, thus the most informative among the instances in the unlabeled pool. The incorrect labeling of these critical samples may produce a notable bias in the learning model, thus affecting the end results significantly. However, despite that, the active learners still show significant improvement and good learning output even in the extreme case of the 25% wrong labeled samples. More interestingly, the graphs show that when the seed size is relatively large, the models are remarkably resilient to the 5% wrong label probability, being able to provide similar results to the 0% error labeling approach after 1000 queries. Based on these graphs, the impact of a wrong labeling probability below 15% might be tolerable and yield good performance over time. A higher probability of mistakes (e.g., 25%) may affect significantly the performance of the active learners and passive models.

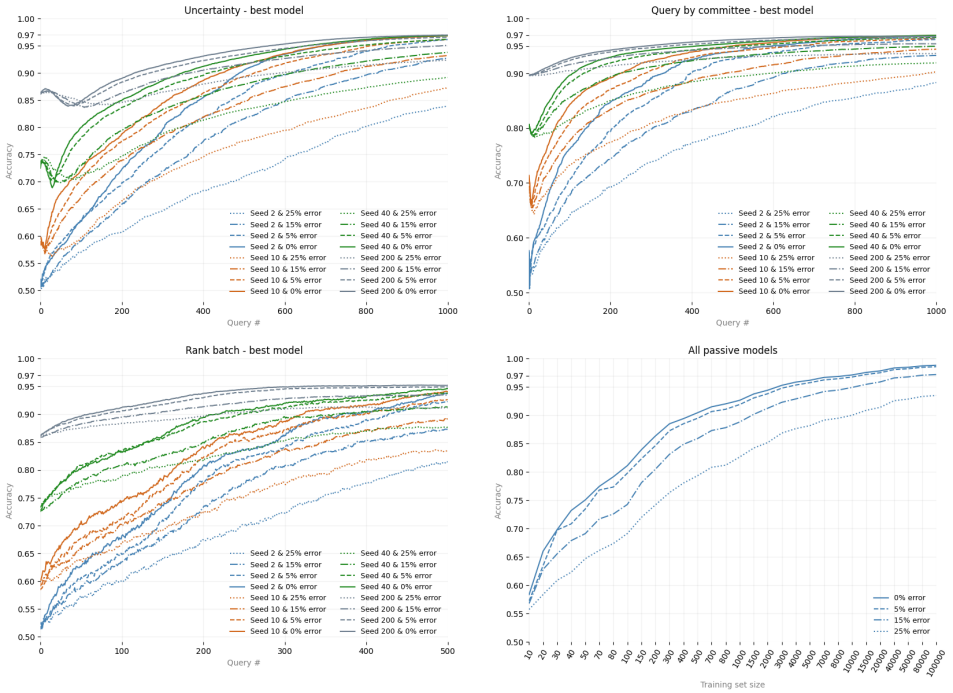


Figure 61: Wrong labeling impact [45]

13.3 Chapter summary

This chapter evaluated different ML-based approaches for early IoT botnet detection using *MedBIoT* data set. Supervised and unsupervised ML models were induced, reporting high-performance metrics. Besides, the active learning approach was evaluated using different strategies and scenarios. The *active learners* were able to provide high-performance metrics with significantly fewer labeled data than the passive approaches. The reported results evidence the benefits of the active learning approach to minimize the labeling cost for those environments, such as SOCs, where a wealth of data is available, but data labeling is expensive and time-consuming.

14 Conclusions and future work

This doctoral dissertation tackles existing challenges and research gaps related to the design and effectiveness of machine learning-based Android malware detection systems such as concept drift and cross-device detection issues. Besides, it addresses the impact of feature selection methods on IoT botnet attack detection models and explores the generation of ML-based early IoT botnet detection systems for the prevention of cyber attacks. Challenges that have been overlooked by specialized research. The following subsections detail the main contributions and novelty of this doctoral thesis in the light of the initial research objectives (RO).

14.1 Android malware detection

Android users are under siege as the popular OS is the most targeted mobile platform by malicious actors. The open-source nature of the OS and large target audience make the popular OS an enticing objective for cyber attackers whose most common motivation is financial revenue. However, when a piece of malware infects an Android device, it can be used not only for such a direct economic purpose (e.g., sending premium SMS) but also to steal, hijack, or corrupt the wealth of sensitive data stored in these portable devices. Aiming to protect vulnerable end-users, Android malware detection research has focused on the application of machine learning methods as a means to overcome the limitation of the traditional AV approaches in the mobile platform. Part I of this dissertation tackles issues related to Android malware detection research.

In this regard, one of the major contributions of this work to the Android malware research domain is the generation of *KronoDroid*, a novel, publicly available data set that enables the study of concept drift and cross-device detection issues (RO1). *KronoDroid* data set is the cornerstone of this research as it enabled all the subsequent investigations performed. More specifically, *KronoDroid* main features were leveraged to perform a thorough exploration of concept drift issues in two of the most commonly used feature spaces for Android research, system calls and permissions (RO2), and propose a method based on a data stream approach to handle concept drift effectively and maintain high detection performance metrics over time (RO3). Further, the inner workings of the adaptive solution proposed to handle concept drift were leveraged to characterize the phenomenon in combination with interpretation techniques (RO2). The proposed solution was then used to explore the impact of the timestamps provided by *KronoDroid* in concept drift modeling and handling (RO4). A typical approach to update the detection model knowledge and address data evolution issues in production setups is model retraining. In this regard, the active learning approach was explored as an alternative means to address concept drift while minimizing the data labeling needs for model retraining (RO3). In addition to the *temporal* dimension, *KronoDroid* includes the data *source* perspective, enabling the study of behavioral differences among Android platforms (i.e., real devices and emulators). In this regard, it enabled us to explore cross-device detection issues under data evolution constraints (RO5). Finally, the design and implementation of a thorough benchmarking enabled us to assess the validity of the cross-device behavioral postulate (RO5).

As a result, the research outcomes and main findings of *Part I* of this thesis, detailed in each chapter, address existing challenges in Android malware detection overlooked by the specialized research and contribute to enhancing the effectiveness and efficiency of the ML-based Android detection systems. Even though the research is far from complete and the aspects investigated in this work may still be understood as open challenges, the author of this thesis would be pleased if the work performed in this problem domain can

inspire or help others to consider and address these issues in the design and implementation of better detection systems, and make the digital world safer.

14.2 IoT botnet detection

IoT botnet attacks can have nefarious consequences for individuals and companies. From financial losses to a severe impact on the reputation and consequent loss of trust from current and potential customers, these massive-scale attacks constitute a potentially devastating threat. As a result, most IoT botnet detection-related research has focused on attack detection. In this context, the first chapter of Part II of this dissertation explores the benefits of feature selection techniques to induce better attack detection systems (RO6). Besides, the relation between these dimensionality reduction techniques and post-hoc interpretation methods in the generation of decision explanations, which can help the experts to understand the attacks, is investigated (RO6).

However, even though an enhanced attack detection can help to mitigate the consequences of large-scale attacks; a more interesting issue is attack prevention. In this respect, early IoT botnet detection can potentially help to dismantle the botnet formation efforts and prevent the generation of the attacks. The second chapter of Part II of this thesis explores this topic. More specifically, *MedBIoT* data set is used to induce effective supervised and unsupervised ML-based detection models that can discriminate effectively between malicious and benign IoT botnet traffic in the early stages of botnet formation (RO7). For the same purpose, the active learning approach is evaluated in the context of SOCs, where this learning strategy can help to induce effective detection models that minimize the data labeling needs and, consequently, reduce the expensive cost associated with the labeling effort (RO8).

The main findings of Part II of this thesis, detailed in each chapter, tackled relevant aspects to enhance the effectiveness and efficiency of the machine learning-based IoT botnet detection systems. Currently, there are many excellent researchers focused on this task in this complex and evolving field, thus our work can be understood as a small contribution to the problem domain. Our aim will be fulfilled if our results are considered in the design and induction of IoT botnet detection systems.

14.3 Limitations and threats to validity

Machine learning-based systems rely on data quantity and, most importantly, data quality for their success at a given task (e.g., malware detection). Besides, the methodology followed and techniques and algorithms used can significantly impact the obtained outcomes. Consequently, the threats to the validity of our results and the limitations of this research arise from two main sources: the data used and the approaches and techniques employed.

The representativeness of the data is a central element for machine learning-based systems and data analysis. This dissertation is based on two data sets: *KronoDroid* and *MedBIoT*. Regarding *KronoDroid*, despite being one of the largest publicly available Android data sets and providing samples for an extended historical period, the representativeness of the data set is not guaranteed. The Android threat landscape is complex and ever-evolving, with thousand of malicious applications discovered monthly. Therefore, only sampling-based approximations of the dynamic malware threat landscape at a specific period can be achieved. In this regard, given the large size of *KronoDroid* and the combination of data sources, the representativeness of the data set is maximized but not ensured. In this regard, *KronoDroid* could benefit from integrating more data samples

(i.e., especially from recent years) that could enlarge the representativeness of the data set and, consequently, increase the internal validity of the results. The same reasoning applies to the *MedBIoT* data set.

The techniques and algorithms employed for data analysis, concept drift handling, and characterization are also not free from limitations and assumptions. For example, the data stream approach for concept drift handling assumes that timestamps are accessible and valid for managing the phenomenon effectively. However, as demonstrated, timestamps might not always be accessible or valid. A larger data set could mitigate this issue. Besides, the algorithms used to induce the classification models can have a significant impact on the detection performance. Our methodology included testing several classification algorithms to select the best performer for our solution. However, algorithmic enhancement of our solution cannot be discarded, given the fast pace of development of new algorithms and improvement of the existing ones by the machine learning community. The characterization techniques employed and data analysis tools utilized in this research (i.e., feature selection methods and statistical metrics and tests) are widely used and accepted scientific methods. However, they usually make assumptions about the data that may impact the analysis and outcomes. For that purpose, different techniques have been used in this research, enabling the exploration of the phenomenon from different viewpoints (e.g., Shapley values, permutation feature importance, Fisher's score) and providing complementary analytical approaches to the studied phenomenon (e.g., important features for concept drift). The consistency of the results obtained using the different techniques indicates that the results are solid. However, the usage of additional approaches and techniques may enrich the analysis and also contrast the obtained results.

14.4 Future work

The research presented in this doctoral dissertation explored two application domains where machine learning can help significantly to address complex challenges in ever-evolving data scenarios. Consequently, the work tackled in this research is far from complete and most probably, will never be. However, there are several aspects that could be improved and explored further in future research.

For instance, the generation of *KronoDroid*-like data sets, thus including temporal and device-related aspects, could not only be used to contrast the findings of this work but also to complement it. The inclusion of more recent samples, different timestamps, data from more devices, etc., could assist in devising enhanced detection systems for malware detection. This would also enable further exploration of the issues introduced in this thesis such as concept drift and cross-device data detection, but also other topics that were briefly surfaced such as malware family evolution and multi-class detection.

The adaptive solution proposed in this work was capable of handling concept drift effectively in the analyzed period. However, as with any algorithm, it might be subject to improvements, especially to become more robust against adversarial drift attacks. In these attacks, the attacker induces an artificial drift in the data to *fool* the classifier, thus provoking adaptation to false data and harming the detection capabilities of the classifier. Consequently, the exploration of adversarial drift is one of our future research paths in the domain.

This work also opened the path of early IoT botnet detection that may assist in attack prevention. Our work and its related data set *MedBIoT* may be seen as an initial exploration of the phenomenon that may foster future research in this direction.

Data quality and quantity are critical factors that empower machine learning systems to excel in recognition tasks. Consequently, the focus on data set generation can open

significant research paths and lead to substantial enhancements in the related detection systems for the years to come. Besides, the exploration of algorithmic improvements to existing methods and application of new machine learning or deep learning approaches can be of great benefit to foster research and, ultimately, contribute to a safer *digital* future.

List of Figures

List of Tables

References

[1] C. C. Aggarwal. *Data mining: the textbook*. Springer, 2015.

[2] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Are your training datasets yet relevant? In *International Symposium on Engineering Secure Software and Systems*, pages 51–67. Springer, 2015.

[3] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer. Emulator vs real phone: Android malware detection using machine learning. In *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics*, IWSPA '17, page 65–72, New York, NY, USA, 2017. Association for Computing Machinery.

[4] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer. Emulator vs real phone: Android malware detection using machine learning. In *Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics*, pages 65–72, 2017.

[5] M. K. Alzaylaee, S. Y. Yerima, and S. Sezer. Dl-droid: Deep learning based android malware detection using real devices. *Computers & Security*, 89:101663, 2020.

[6] M. R. Amin, M. Zaman, M. S. Hossain, and M. Atiquzzaman. Behavioral malware detection approaches for android. In *2016 IEEE International Conference on Communications (ICC)*, pages 1–6, 2016.

[7] Android. Run apps on the android emulator. `https://developer.android.com/studio/run/emulator`, 2021.

[8] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al. Understanding the mirai botnet. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

[9] APKMirror. Apkmirror. `https://www.apkmirror.com/`, 2020.

[10] ArgusLab. Amd dataset - argus cyber security lab. `http://amd.arguslab.org/`, 2020.

[11] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[12] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[13] AV-Test. Malware. https://www.av-test.org/en/statistics/malware/, 2021.

[14] AV-Test. Development of android malware. `https://www.av-test.org/en/statistics/malware/`, 2022.

[15] H. Bahşi, S. Nõmm, and F. B. La Torre. Dimensionality reduction for machine learning based iot botnet detection. In *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 1857–1862, 2018.

[16] F. Barbero, F. Pendlebury, F. Pierazzi, and L. Cavallaro. Transcending transcend: Revisiting malware classification with conformal evaluation. *arXiv preprint arXiv:2010.03856*, 2020.

[17] V. H. Bezerra, V. G. T. da Costa, R. A. Martins, S. B. Junior, R. S. Miani, and B. B. Zarpelao. Data set. `http://www.uel.br/grupo-pesquisa/secmq/dataset-iot-security.html`, 2018.

[18] L. Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[19] N. Buchka and M. Kuzin. Attack on zygote: a new twist in the evolution of mobile threats. https://securelist.com/attack-on-zygote-a-new-twist-in-the-evolution-of-mobile-threats/74032/, 2016.

[20] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26, 2011.

[21] H. Cai. Assessing and improving malware detection sustainability through app evolution studies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–28, 2020.

[22] H. Cai, N. Meng, B. Ryder, and D. Yao. Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security*, 14(6):1455–1470, 2019.

[23] R. Casolare, C. De Dominicis, G. Iadarola, F. Martinelli, F. Mercaldo, and A. Santone. Dynamic mobile malware detection through system call-based image representation. *J. Wirel. Mob. Networks Ubiquitous Comput. Dependable Appl.*, 12(1):44–63, 2021.

[24] M. Dimjašević, S. Atzeni, I. Ugrina, and Z. Rakamaric. Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 1–8, 2016.

[25] R. Doshi, N. Apthorpe, and N. Feamster. Machine learning ddos detection for consumer internet of things devices. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 29–35. IEEE, 2018.

[26] Dr.Web. Doctor web: banking trojan android.bankbot.149.origin has become a rampant tool of cybercriminals. https://news.drweb.com/show/?i=11772, 2018.

[27] U. du Luxembourg. Androzoo - lists of apks. `https://androzoo.uni.lu/lists`, 2021.

[28] F-droid. F-droid - free and open source android app repository. `https://f-droid.org/`, 2020.

[29] F-secure. Trojan:android/droiddream.a. https://www.f-secure.com/v-descs/trojan_android_droiddream_a.shtml, 2021.

[30] F-secure. Trojan:android/geinimi. https://www.f-secure.com/v-descs/trojan_android_geinimi.shtml, 2021.

[31] P. Feng, J. Ma, C. Sun, X. Xu, and Y. Ma. A novel dynamic android malware detection system with ensemble learning. *IEEE Access*, 6:30996–31011, 2018.

[32] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):1–37, 2014.

[33] Google. Android market: Now available for users. https://android-developers.googleblog.com/2008/10/android-market-now-available-for-users.html, 2008.

[34] Ö. Gözüaçık and F. Can. Concept learning using one-class classifiers for implicit drift detection in evolving data streams. *Artificial Intelligence Review*, (0123456789), 2020.

[35] A. Guerra-Manzanares and H. Bahsi. On the application of active learning for efficient and effective early iot botnet detection. *Journal article*, *under review*, 2022.

[36] A. Guerra-Manzanares and H. Bahsi. On the application of active learning to handle data evolution in android malware detection. *Conference paper*, *under review*, 2022.

[37] A. Guerra-Manzanares and H. Bahsi. On the relativity of time: Implications and challenges of data drift on long-term effective android malware detection. *Computers & Security*, *in press*:102835, 2022.

[38] A. Guerra-Manzanares, H. Bahsi, and M. Luckner. Leveraging the first line of defense: A study on the evolution and usage of android security permissions for enhanced android malware detection. *Journal of Computer Virology and Hacking Techniques*, *in press*, 2022.

[39] A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Differences in android behavior between real device and emulator: A malware detection perspective. In *Proceedings of the 6th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 399–404, 2019.

[40] A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Hybrid feature selection models for machine learning based botnet detection in iot networks. In *2019 International Conference on Cyberworlds (CW)*, pages 324–327, 2019.

[41] A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Kronodroid: Time-based hybrid-featured dataset for effective android malware detection and characterization. *Computers & Security*, 110:102399, 2021.

[42] A. Guerra-Manzanares, M. Luckner, and H. Bahsi. Android malware concept drift using system calls: Detection, characterization and challenges. *Expert Systems with Applications*, 206:117200, 2022.

[43] A. Guerra-Manzanares, M. Luckner, and H. Bahsi. Concept drift and cross-device behavior: Challenges and implications for effective android malware detection. *Computers & Security*, 120:102757, 2022.

[44] A. Guerra-Manzanares, J. Medina-Galindo, H. Bahsi, and S. Nõmm. Medbiot: Generation of an iot botnet dataset in a medium-sized iot network. In *Proceedings of the 6th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*, pages 207–218, 2020.

[45] A. Guerra-Manzanares, J. Medina-Galindo, H. Bahsi, and S. Nõmm. Using medbiot dataset to build effective machine learning-based iot botnet detection systems. In *International Conference on Information Systems Security and Privacy*, pages 222–243. Springer, 2020.

[46] A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. In-depth feature selection and ranking for automated detection of mobile malware. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*, pages 274–283. INSTICC, SciTePress, 2019.

[47] A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. Time-frame analysis of system calls behavior in machine learning-based mobile malware detection. In *International Conference on Cyber Security for Emerging Technologies (CSET)*, pages 1–8, 2019.

[48] A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. Towards the integration of a post-hoc interpretation step into the machine learning workflow for iot botnet detection. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 1162–1169, 2019.

[49] A. Guerra-Manzanares and M. Välbe. Cross-device behavioral consistency: Benchmarking and implications for effective android malware detection. *Machine Learning with Applications*, 9:100357, 2022.

[50] N. Hachem, Y. B. Mustapha, G. G. Granadillo, and H. Debar. Botnets: lifecycle and taxonomy. In *2011 Conference on Network and Information Systems Security*, pages 1–8. IEEE, 2011.

[51] Q. Han, V. S. Subrahmanian, and Y. Xiong. Android malware detection via (somewhat) robust irreversible feature transformations. *IEEE Transactions on Information Forensics and Security*, 15:3511–3525, 2020.

[52] G. Iadarola, F. Martinelli, F. Mercaldo, and A. Santone. Towards an interpretable deep learning model for mobile malware detection and family identification. *Computers and Security*, 105:102198, 2021.

[53] IBM. Overfitting. `https://www.ibm.com/cloud/learn/overfitting`, 2021.

[54] P. Irolla and A. Dey. The duplication issue within the drebin dataset. *Journal of Computer Virology and Hacking Techniques*, 14(3):245–249, 2018.

[55] M. Jerbi, Z. C. Dagdia, S. Bechikh, and L. B. Said. On the use of artificial malicious patterns for android malware detection. *Computers & Security*, 92:101743, 2020.

[56] X. Jiang and Y. Zhou. *Android malware*. Springer, 2013.

[57] J. Johnson. Development of new android malware worldwide from june 2016 to march 2020. https://www.statista.com/statistics/680705/global-android-malware-volume/, 2021.

[58] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 625–642, 2017.

[59] H. Kang, D. H. Ahn, G. M. Lee, J. D. Yoo, K. H. Park, and H. K. Kim. Iot network intrusion dataset. `http://dx.doi.org/10.21227/q70p-q449`, 2019.

[60] R. R. Karn, P. Kudva, H. Huang, S. Suneja, and I. M. Elfadel. Cryptomining Detection in Container Clouds Using System Calls and Explainable Machine Learning. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):674–691, 2021.

[61] M. Kinkead, S. Millar, N. McLaughlin, and P. O'Kane. Towards explainable cnns for android malware detection. *Procedia Computer Science*, 184(2019):959–965, 2021.

[62] N. Kiss, J.-F. Lalande, M. Leslous, and V. V. T. Tong. Kharon dataset: Android malware under a microscope. In *The {LASER} Workshop: learning from authoritative security experiment results ({LASER} 2016)*, pages 1–12, 2016.

[63] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[64] N. Koroniotis, N. Moustafa, E. Sitnikova, and B. Turnbull. Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset. *Future Generation Computer Systems*, 100:779–796, 2019.

[65] T. Lei, Z. Qin, Z. Wang, Q. Li, and D. Ye. Evedroid: Event-aware android malware detection against model degrading for iot devices. *IEEE Internet of Things Journal*, 6(4):6668–6680, 2019.

[66] J. Leonard, S. Xu, and R. Sandhu. A framework for understanding botnets. In *2009 International Conference on Availability, Reliability and Security*, pages 917–922. IEEE, 2009.

[67] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai. Identifying android malicious repackaged applications by thread-grained system call sequences. *computers & security*, 39:340–350, 2013.

[68] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 422–433, 2015.

[69] R. Lipovsky, L. Stefanko, and G. Branisa. Trends in android ransomware. https://www.welivesecurity.com/wp-content/uploads/2017/02/ESET_Trends_2017_in_Android_Ransomware.pdf, 2017.

[70] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu. A review of android malware detection approaches based on machine learning. *IEEE Access*, 8:124579–124607, 2020.

[71] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, and G. Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31(12):2346–2363, 2018.

[72] A. Margara and T. Rabl. *Definition of Data Streams*, pages 1–4. Springer International Publishing, Cham, 2018.

[73] A. Marzano, D. Alexander, O. Fonseca, E. Fazzion, C. Hoepers, K. Steding-Jessen, M. H. Chaves, Í. Cunha, D. Guedes, and W. Meira. The evolution of bashlite and mirai iot botnets. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00813–00818. IEEE, 2018.

[74] C. D. McDermott, F. Majdani, and A. V. Petrovski. Botnet detection in the internet of things using deep learning approaches. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2018.

[75] J. A. Medina Galindo. Generation of malware behavioral datasets in a medium scale iot network. 2019.

[76] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, A. Shabtai, D. Breitenbacher, and Y. Elovici. N-baiot—network-based detection of iot botnet attacks using deep autoencoders. *IEEE Pervasive Computing*, 17(3):12–22, 2018.

[77] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81, 1956.

[78] A. Narayanan, L. Yang, L. Chen, and L. Jinliang. Adaptive and scalable android malware detection through online learning. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pages 2484–2491, 2016.

[79] S. Nõmm and H. Bahşi. Unsupervised anomaly based botnet detection in iot networks. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 1048–1053, 2018.

[80] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security (TOPS)*, 22(2):1–34, 2019.

[81] A. Parmisano, S. Garcia, and M. J. Erquiaga. Stratosphere laboratory. a labeled dataset with malicious and benign iot network traffic. `https://www.stratosphereips.org/datasets-iot23`, 2020.

[82] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro. {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 729–746, 2019.

[83] S. Ramírez-Gallego, B. Krawczyk, S. García, M. Woźniak, and F. Herrera. A survey on data preprocessing for data stream mining: Current status and future directions. *Neurocomputing*, 239:39–57, 2017.

[84] M. T. Ribeiro, S. Singh, and C. Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[85] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, 15(1):83–97, 2018.

[86] M. Scalas, D. Maiorca, F. Mercaldo, C. A. Visaggio, F. Martinelli, and G. Giacinto. On the effectiveness of system API-related information for Android ransomware detection. *Computers and Security*, 86:168–182, 2019.

[87] T. Seals. Slocker android ransomware resurfaces in undetectable form. `https://www.infosecurity-magazine.com/news/slocker-android-ransomware/`, 2017.

[88] B. Settles. Active learning literature survey. 2009.

[89] B. Settles and M. Craven. An analysis of active learning strategies for sequence labeling tasks. In *proceedings of the 2008 conference on empirical methods in natural language processing*, pages 1070–1079, 2008.

[90] T. Sharma and D. Rattan. Malicious application detection in android — a systematic literature review. *Computer Science Review*, 40:100373, 2021.

[91] M. Shipman. More bad news: Two new pieces of android malware – plankton and yzhcsms. https://news.ncsu.edu/2011/06/wms-android-plankton/, 2011.

[92] R. Shire, S. Shiaeles, K. Bendiab, B. Ghita, and N. Kolokotronis. Malware squid: A novel iot malware traffic analysis framework using convolutional neural network and binary visualisation. In *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, pages 65–76. Springer, 2019.

[93] S. S. Silva, R. M. Silva, R. C. Pinto, and R. M. Salles. Botnets: A survey. *Computer Networks*, 57(2):378–403, 2013.

[94] R. Surendran, T. Thomas, and S. Emmanuel. Gsdroid: Graph signal based compact feature representation for android malware detection. *Expert Systems with Applications*, 159:113581, 2020.

[95] A. Toh. Azure ddos protection—2021 q3 and q4 ddos attack trends. https://azure.microsoft.com/en-us/blog/azure-ddos-protection-2021-q3-and-q4-ddos-attack-trends/, 2021.

[96] J. M. Vidal, A. L. S. Orozco, and L. G. Villalba. Malware detection in mobile devices by analyzing sequences of system calls. *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 11(5):594–598, 2017.

[97] P. Vinod, A. Zemmari, and M. Conti. A machine learning based approach to detect malicious android apps using discriminant system calls. *Future Generation Computer Systems*, 94:333–350, 2019.

[98] VirusShare. Virusshare. `https://virusshare.com/`, 2020.

[99] VirusTotal. Virustotal academic malware samples. `http://www.virustotal.com`, 2020.

[100] X. Wang and C. Li. Android malware detection through machine learning on kernel task structures. *Neurocomputing*, 435:126–150, 2021.

[101] S. Weagle. Financial impact of mirai ddos attack on dyn revealed in new data. Retrieved from: `https://www.corero.com/blog/797-financial-impact-of-mirai-ddos-attack-on-dyn-revealed-in-new-data.html`, 2017.

[102] G. I. Webb, R. Hyde, H. Cao, H. L. Nguyen, and F. Petitjean. Characterizing concept drift. *Data Mining and Knowledge Discovery*, 30(4):964–994, 2016.

[103] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 252–276. Springer, 2017.

[104] W. Wei, J. Wang, Z. Yan, and W. Ding. Epmdroid: Efficient and privacy-preserving malware detection based on sgx through data fusion. *Information Fusion*, 2022.

[105] S. Weisman. Emerging threats - what is a distributed denial of service attack (ddos) and what can you do about them? Retrieved from: `https://us.norton.com/internetsecurity-emerging-threats-what-is-a-ddos-attack-30sectech-by-norton.html`, 2019.

[106] L. Wu. Android mobile ransomware: Bigger, badder, better? $\texttt{https://www.trendmicro.com/en}_{u}s/research/17/h/android-mobile-ransomware-evolution.html, 2017.$

[107] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah. Android malware detection based on system call sequences and lstm. *Multimedia Tools and Applications*, 78(4):3979–3999, 2019.

[108] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu. Droidevolver: Self-evolving android malware detection system. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 47–62. IEEE, 2019.

[109] R. Yu. Ginmaster : a case study in android malware. https://www.virusbulletin.com/conference/vb2013/abstracts/ginmaster-case-study-android-malware, 2013.

[110] B. B. Zarpelão, R. S. Miani, C. T. Kawakani, and S. C. de Alvarenga. A survey of intrusion detection in internet of things. *Journal of Network and Computer Applications*, 84:25–37, 2017.

[111] N. Zhang, J. Xue, Y. Ma, R. Zhang, T. Liang, and Y.-a. Tan. Hybrid sequence-based android malware detection using natural language processing. *International Journal of Intelligent Systems*, 36(10):5770–5784, 2021.

[112] X. Zhang, Y. Zhang, M. Zhong, D. Ding, Y. Cao, Y. Zhang, M. Zhang, and M. Yang. Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 757–770, 2020.

[113] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

[114] P. Zyblewski, R. Sabourin, and M. Woźniak. Preprocessed dynamic classifier ensemble selection for highly imbalanced drifted data streams. *Information Fusion*, 66(June 2020):138–154, 2021.

Acknowledgements

I would like to extend my gratitude to a variety of individuals who helped me to write this dissertation, for although I am the author, no man is an island, and I could not have done this work alone. I owe this research and its outcomes to the many people who have influenced it, either directly or indirectly. In this regard, I would like to thank my family, supervisors, friends, and colleagues for your continued support and help along the road. I could not have reached the end line alone.

I would like to give special thanks to those whose contribution to this thesis has been even more significant and crucial at some point during this four-year-long journey. I would like to offer my warmest thanks to my beloved mother, Fina, who has always shown so much love and support during my life journey, no matter how far life has brought me from her or how much she still suffers and cries in our farewells. *Gracias por darme todo mama*. I would like to thank and dedicate this dissertation to my *yayo*, who, despite never comprehending why I was always so far from home if I could work in my hometown, always showed me unconditional love and taught me the value of effort and how a real gentleman behaves through example. *Allí donde estés*, *gracias por cuidar de mi yayo*, *te echo de menos*. To my dear Duska, who has supported me along this journey, showing so much love, respect, and understanding like no one would ever do. *Grazie mille* for being my biggest supporter and standing by my side during this stressful period. To my (step)father, Francesc, for the care and support all these years. *Moltes gràcies per tot*. I would also like to extend my thanks to my main supervisor, Hayretdin, who has taught me so many things during my research path that I could fill another dissertation (or a couple more); *teşekkürler* for teaching me so much and especially for answering the email that started this project and for helping that random exchange student asking for free supervision to achieve a BSc, MSc, and now a Ph.D. degree. To Risto, *suur aitäh* for your kindness, for counting on me for many projects, for always asking and respecting my opinion, and for giving this dissertation a proper Estonian title and abstract. To Marcin, *dziękuję* for always finding time for me in your busy schedule to help me in a critical time during my Ph.D. studies.

Even though this is not the best doctoral dissertation ever written, it is my dissertation, and I cannot be more proud and glad to reach the end line accompanied by all of you. Thank you for making this achievement possible.

Abstract

Machine Learning-Based Detection and Characterization of Evolving Threats in Mobile and IoT Systems

This dissertation explores the application of machine learning methods to overcome the present challenges that affect two significant research areas in the cyber security domain: *Android* malware detection and *Internet of Things* (IoT) botnet detection.

The majority of Android malware detection solutions neglect the impact of concept drift on the performance of the detection models over time. This dissertation explores the phenomenon by demonstrating, addressing and characterizing concept drift in the most commonly used Android feature spaces to induce detection models (i.e., system calls, permissions and API calls). Furthermore, the impact of timestamping approaches in the data modeling and effectiveness of the models is analyzed. Cross-device detection, another challenge for the development of effective detection systems, is comprehensively investigated by examining the validity of the cross-device consistent behavior postulate among Android platforms, which is often assumed in research setups, and its impact on the learning models. The research outcomes, such as the *KronoDroid* data set and the proposed solution to handle concept drift, and main findings of the experimentation performed enable the design of more robust models to *ageing* challenges and the enhancement of Android detection models.

In the IoT domain, the majority of the research studies in the cyber security field focus on attack detection. This doctoral dissertation explores the impact of feature selection on the detection performance and the enhancement of the detection models. Besides, attack prevention, as a means of avoiding the nefarious consequences of IoT-based attacks, is thoroughly explored using supervised and unsupervised machine learning models and the active learning approach. The research outcomes in this domain enable the generation of enhanced detection systems that might be capable of preventing IoT-based attacks and their consequences.

Kokkuvõte

Masinõppepõhine arenevate ohtude tuvastamine ning kirjeldamine mobiilseadmete ja värkvõrkude jaoks

Käesolev väitekiri keskendub masinõppe meetodite kasutamisele, et ületada probleeme kahes olulises küberturvalisuse valdkonnas: Androidi pahavara tuvastamine ja värkvõrgupõhiste botnetide tuvastamine.

Enamik Androidi pahavara tuvastamise lahendustest ei arvesta kontseptuaalse triivi aja jooksul ilmnevat mõju tuvastusmudelite täpsusele. Väitekiri uurib seda nähtust, kirjeldades kontseptuaalset triivi ning pakkudes lahendusi sellega toimetulekuks levinud Androidi tunnusruumide põhjal loodud tuvastusmudelites (süsteemi- ja rakendusliidese funktsioonide väljakutsete ning Androidi õiguste põhised mudelid). Samuti analüüsib väitekiri erinevate ajatembelduse meetodite mõju andmemudelitele ning mudelite efektiivsust. Väitekiri keskendub ka pahavara seadmeülesele tuvastamisele, mis on efektiivsete tuvastussüsteemide loomisel oluline probleem. Väitekirjas käsitletakse pahavara käitumist erinevatel Androidi platvormidel ja varasemates töödes tihti esinevat postulaati, et pahavara käitumine ei sõltu platvormist, ning uuritakse selle mõju masinõppe mudelitele. Uurimistöö tulemused nagu KronoDroid andmekogu, lahendused kontseptuaalse triiviga toimetulekuks ning teised eksperimentide käigus saadud tulemused võimaldavad luua töökindlamaid masinõppe mudeleid.

Varasemad värkvõrkude küberturvalisuse valdkonnas tehtud uurimistööd käsitlevad peamiselt rünnete tuvastamist. Käesolev doktoritöö keskendub tuvastusmudelite tõhustamisele ning sellele, kuidas tunnuste valik mõjutab tuvastuse täpsust. Doktoritöö uurib samuti värkvõrkude vastaste rünnete tõkestamist juhendatud ning juhendamata masinõppe mudelite abil, käsitledes ka aktiivõppe meetodeid. Uurimistöö tulemused võimaldavad luua täiustatud värkvõrkude vastaste rünnete tuvastamise süsteeme, millel on rünnete tõkestamise võimekus, et hoida ära rünnete tekitatud kahju.

Appendix 1

Publication I

A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. In-depth feature selection and ranking for automated detection of mobile malware. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*, pages 274–283. INSTICC, SciTePress, 2019

In-depth Feature Selection and Ranking for Automated Detection of Mobile Malware

Alejandro Guerra-Manzanares, Sven Nõmm and Hayretdin Bahsi
Department of Software Science, TalTech University, Tallinn, Estonia

Keywords: Machine Learning, Mobile Malware, Feature Selection.

Abstract: New malware detection techniques are highly needed due to the increasing threat posed by mobile malware. Machine learning techniques have provided promising results in this problem domain. However, feature selection, which is an essential instrument to overcome the curse of dimensionality, presenting higher interpretable results and optimizing the utilization of computational resources, requires more attention in order to induce better learning models for mobile malware detection. In this paper, in order to find out the minimum feature set that provides higher accuracy and analyze the discriminatory powers of different features, we employed feature selection and ranking methods to datasets characterized by system calls and permissions. These features were extracted from malware application samples belonging to two different time-frames (2010-2012 and 2017-2018) and benign applications. We demonstrated that selected feature sets with small sizes, in both feature categories, are able to provide high accuracy results. However, we identified a decline in the discriminatory power of the selected features in both categories when the dataset is induced by the recent malware samples instead of old ones, indicating a concept drift. Although we plan to model the concept drift in our future studies, the feature selection results presented in this study give a valuable insight regarding the change occurred in the best discriminating features during the evolvement of mobile malware over time.

1 INTRODUCTION

Mobile phone users are increasingly facing the risks of malware. McAfee stated that "2018 could be the year of mobile malware" as they detected 16 million infections in the third quarter of 2017 alone, twice the figure in 2016 (McAfee, 2018). This enormous increase was also confirmed by Kaspersky who identified an 80% rise in mobile malware attacks (Unuchek, 2018). In addition to these spikes, malware detection software has been proved to be inefficient in tackling this threat (Fedler et al., 2013).

Traditional detection approaches based on signatures fail to detect unknown malware due to the improved obfuscation or stealth techniques employed by malware creators (Fedler et al., 2013). On the other side, machine learning techniques have been perceived as a promising approach for detecting previously unseen malware samples and many studies have shown that they could provide high detection accuracy (Sahs and Khan, 2012; Yuan et al., 2014; Arp et al., 2014). These studies created learning models using dynamic, static or both (namely hybrid) features extracted from legitimate applications and malware samples. Static features such as permissions, java codes or intent filters, are extracted directly from APK files whereas dynamic features, e.g. system calls or network traffic patterns, are derived from the interaction of programs with OS or network (Feizollah et al., 2015).

Feature selection, eliminating irrelevant or redundant features that do not improve the classification performance, is an essential step of machine learning workflow due to three reasons: (1) Representing the problem domain with high dimensions requires more data for learning (commonly known as the curse of dimensionality) and may disturb the accuracy of the classifier, (2) Models using higher dimensions cannot be easily interpreted by the experts, which may create enormous problems in detecting falsely classified instances or profoundly investigating a cyber incident, (3) Higher dimensional data requires more computational resources for constructing and using the learning model on a mobile device. On the other side, feature selection could be more complicated in problem domains where the behaviour of the subjects may vary in time, i.e., a selected feature set may no longer have its discriminatory power, which may be one of the main concerns in malware detection.

In this study, our primary objectives are to iden-

Guerra-Manzanares, A., Nõmm, S. and Bahsi, H.
In-depth Feature Selection and Ranking for Automated Detection of Mobile Malware.
DOI: 10.5220/0007349602740283
In *Proceedings of the 5th International Conference on Information Systems Security and Privacy (ICISSP 2019)*, pages 274-283
ISBN: 978-989-758-359-9

tify the minimum feature set that provides higher accuracy, compare the discriminatory powers of feature categories and analyze the results of models induced by datasets belonging to different time-frames. For these purposes, we applied a two-step procedure to the dataset that is composed by system calls (i.e., a dynamic behaviour) and permissions (i.e., a static behaviour), extracted from malware samples and legitimate applications. In the first step, we used statistical hypothesis testing methods to identify the feature set that may have a significant contribution to the classification. In the second step, we employed Fisher's Score and Gini Index which enabled to rank the selected features according to their discriminatory power. We in turn induced machine learning models with different combinations of datasets with varying feature sets. As Android is the most used mobile operating system worldwide, we focused on detection of Android malware (Statista, 2018). For this research, we formed two malware datasets. "Old dataset" which consists of randomly selected apps from Drebin malware dataset, collected between 2010 and 2012 (Arp et al., 2014)."New dataset" formed by randomly choosing samples, belonging to years 2017 and 2018, from VirusTotal Academic malware dataset (VirusTotal, 2018). Third one is called "legitimate dataset" which is composed by benign applications. We utilized various combinations of these datasets for inducing learning models.

This study shows that feature selection and ranking process can significantly reduce the number of features required in a classifier that provides high accuracy for the detection of mobile malware. We found that features possessing most discriminatory power in classification may differ as new malware types evolve over time, indicating a concept drift. Results suggest that behaviour of mobile malware in terms of system calls and permissions has become more similar to legitimate apps over time although there are some variations among the extent of this evolvement in both feature categories.

Our main contribution is a detailed analysis and comparison of feature selection and ranking results obtained for two types of feature categories. One of the distinctive properties of the present paper is that, in addition to the optimization of number of predictors, we analyzed the change in selected features that has occurred due to the evolvement of malware over time.

This paper is organized as follows: Section 2 presents a review of related literature. Method employed in the study is described in Section 3. Results of our experiments are presented and discussed in Section 4 whereas Section 5 concludes the study.

2 LITERATURE REVIEW

Feature selection and ranking methods have been used in various machine learning-based malware detection studies. In Yan et al. (2013) discriminatory power of malware features such as hexdump of binaries, disassembly codes, PE header and system calls are measured by three filter methods, i.e., ReliefF, Chi-squared, F-statistics, and two embedded methods, i.e., L1 regularized methods, L1-logreg and L1-SVC. In this study, it is identified that PE header and system calls are very beneficial to discern malware from legitimate software, and that L1 regularized methods with 100 features provided higher detection rates (Yan et al., 2013). In Ahmadi et al. (2016) discriminatory powers of various static feature categories are measured and compared by using mean decrease impurity notion and random forest classifier in a multi-class malware family classification.

Utilization of feature ranking methods is considerably less common in those studies which provide classifiers specifically for mobile malware detection (Feizollah et al., 2015). Lindorfer et al. (2015) applied Fisher's Score to evaluate the discriminatory power of dynamic and static feature categories. This study found out that required permissions and some dynamic features related to SMS sending and dynamic loading of code have higher discriminatory powers (Lindorfer et al., 2015). Cen et al. (2015), created a classifier using Regularized Logistic Regression with Lasso Norm for source code features (java package, class and function levels). Information Gain, Chi-Square and an embedded method of logistic regression were utilized for feature selection. It was found that 10% of the features selected by Information Gain or Chi-Square are sufficient for high detection rates (Cen et al., 2015). Similarly, in Shabtai et al. (2012) filter methods such as Chi-Square, Fisher's Score and Information Gain were applied to some system metric features (e.g., CPU consumption, number of running processes, battery level) in the early times of Android.

Pehlivan et al. (2014) applied feature selection methods such as Information Gain, ReliefF, Correlation Feature Selection (CFS) and consistency-based selection to permissions with different classification models. Random forest classifier that selected 25 permission features with CFS provided the best accuracy. In a similar study by Nezhadkamali et al. (2017), three feature selection methods, L1-based feature selection, Information Gain and Gini Impurity, were used with permissions. All three methods were tested using different machine learning algorithms, such as decision tree, SVM and Random forest. Best results

were obtained using Random Forest as classification algorithm and Information Gain as feature selection method (Nezhadkamali et al., 2017).

Sing and Hofmann (2017) used three feature selection methods (Chi-Square, Information gain, and correlation analysis) to select variables and form system calls vector. In Ferrante et al. (2016), an embedded feature selection method was used for classifying the dataset that consisted of features such as system calls, memory usage and CPU usage. Kim and Choi (2014) used Linux kernel features related to memory, CPU and network (summing up to 59 features) to perform malware detection. This study used an embedded model to perform feature selection, ending up eliminating 23 features and using 36 features for their detection system (Kim and Choi, 2014). In Qiao et al. (2016) combined API calls and permissions were processed by two feature selection methods, one-way analysis of variance (ANOVA) (i.e., a filter method) and Support Vector Machine—Recursive Feature Elimination (i.e., a wrapper method). They ended up with top 300 features from API set and 80 from permissions set (Qiao et al., 2016).

Although previously mentioned studies applied feature selection methods and some of them provided considerably detailed analysis about discriminatory powers of used features, none of them analyses the character change and its impact on feature selection.

In Hu et al. (2017) concept drift of mobile malware was modelled with an ensemble learning model in which the feature selection is based on Information Gain. In Jordaney et al. (2017) a concept drift detection method that was based on conformal evaluator is applied to two cases, a binary classification for mobile malware and a multi-class classification for malware. These studies focus on enhancing the detection performance of classifiers with concept drift. However, they do not provide an in-depth analysis of discriminatory powers of feature categories and their impact on concept drift.

3 METHOD

We formulated mobile malware detection as a binary classification problem that requires the discrimination of benign mobile applications from mobile malware samples. As we were able to obtain labelled data, supervised machine learning methods were applied. We followed machine learning workflow, that mainly involves five steps: (1) Data Acquisition, (2) Data Cleaning and Preparation, (3) Feature Selection, (4) Classifier training and Evaluation, (5) Interpretation (Robert, 2014). Sometimes tuning could be applied to the trained classifier, but within the framework of the present study, this step was omitted as it was deemed as unnecessary.

We tested *k*-nearest neighbours (kNN), logistic regression, decision tree, and support vector machines (SVM) for building the classifiers, and used Python programming language and Sci-kit learn library in our implementation. Data acquisition and feature selection stages are detailed in Sections 3.1 and 3.2. We covered two types of feature categories in our datasets: absolute frequency of system calls (numerical features) encountered during the execution of the applications and requested Android standard permissions (categorical features).

3.1 Data Acquisition

In this study, we collected 3000 Android x86 architecture compatible applications as the details are given below:

- 1000 benign applications which were randomly downloaded by the authors from APKMirror repository. They were verified as malware free applications with VirusTotal AntiVirus engine. Legitimate applications date between April 2017 and February 2018. Named as ”legitimate dataset” in this research.
- 1000 malware applications which were randomly selected from Drebin malware dataset. These samples date between August 2010 and October 2012 (Arp et al., 2014). We named this dataset as ”old malware dataset”, and refer to each element in the set as ”old malware”.
- 1000 malware applications which were randomly selected from VirusTotal Academic malware dataset. This dataset, shared by VirusTotal, dates between the end of 2016 and beginning 2018 (VirusTotal, 2018). We named this dataset as ”new malware dataset”, and refer to each element in the set as ”new malware”.

Android requested permissions were directly extracted from AndroidManifest.xml file, included in every application APK file, using Android Asset Packaging Tool (*aapt*). The recent Android distribution, Android 8.0, defines 147 Android standard permissions. A permission profile vector that is composed of the data regarding the presence/absence of each Android standard permission was created for each application.

As the collection of system calls requires to run the application itself, we used an Android emulation environment and Android Debug Bridge (*ADB*) to install, execute, monitor, log and uninstall each applica-

tion. During the execution, *strace* tool was attached to the main process to obtain the first 2000 system calls. 212 distinct system calls are defined in Bionic x86 library. A frequency vector that included the number of each system call made by the application was formed from the logged data. Prior research have demonstrated that malware could be effectively discriminated with a reduced amount of system calls acquired during the application's boot up and that acquisition of the first 2000 system calls provided the best detection results (Vidal et al., 2017).

Although we selected malware samples from two different time-frames, composing two different malware datasets, we used only one benign dataset comprised of recent applications. In this study, we focused on the analysis of change in selected features according to the evolvement of malware with respect to recent benign applications. This approach is in line with malware detection practices happening in the field as mobile phones are usually not compatible with older applications due to frequent operating system and hardware changes and also changes in applications' installation requirements but the detection systems usually include signatures of all malware samples including the old ones. The impact of the evolvement in benign applications will also be analyzed in the context of concept drift within our future studies.

3.2 Feature Selection and Ranking

We employed a two-step procedure that consists of conducting statistical hypothesis testing for feature selection and applying feature ranking method. The former one chooses the features which significantly differ between the two classes (i.e., legitimate and malware), and the latter one orders the features according to their discriminatory power. Order provided in this step is necessary to optimize the number of features used as predictors and describe behavioural evolvement of malware belonging to different timeframes.

There are three feature selection techniques that can be widely utilized in identifying the features (Aggarwal, 2015). Filter techniques evaluate the suitability of a feature by using a statistical criterion which can be applied irrespective of the classification method used. Wrapper techniques iteratively extend the feature set and evaluate the accuracy of each identified set in a classification model. Embedded techniques also evaluate suitability of the feature set with respect to a particular classification model, but unlike the wrapper one, they attempt to prune the features within the classification process itself. Since wrapper and embedded techniques have higher computational complexity, we utilized filter techniques in the second step.

It is important to emphasize that feature categories used in this study, system calls and permissions, do not have the same data type. System calls are numeric values (i.e., amount of calls issued for each system call) and permissions are categorical (i.e., permission request was present/absent for each standard permission). In both steps, we employed different techniques that are more appropriate for each feature category and its data type. The procedure was performed as follows:

- Step 1: Feature selection by statistical hypothesis testing
  - **System Calls.** System calls which differ between malicious and legitimate applications in terms of mean values were selected. To perform statistical hypothesis testing Welch's Test was used. This test provides more reliable results for the cases of unequal variances (Welch, 1947). The statement of the null (base) hypothesis H_o is that mean values of for the number of system calls among first 2000 calls are the same for legitimate μ_L and malicious μ_M applications, and the statement of the alternative hypothesis H_1 is that mean values are different.

$$H_0: \quad \mu_L = \mu_M$$
$$H_1: \quad \mu_L \neq \mu_M$$

  - **Permissions.** As these features are categorical, we employed χ^2 (chi-squared independence test) which can answer the question if two categorical variables are related or not. The statement of the null hypothesis is that there is no relation between the particular permission and class of the application. The statement of the alternative hypothesis is that there is a relation between particular permission and class.
- Step 2. Feature ranking by Fisher's Score and Gini Index
  - **System Calls.** Fisher's Scores of system calls with mean values that differ significantly between malicious and legitimate applications were computed (i.e., higher Fisher's score values indicate higher discriminatory power).
  - **Permissions.** As permissions are categorical, Gini Index suited better for ordering these features (i.e., lower values of the Gini Index indicate higher discriminatory power).

At first glance, a two step procedure may seem unnecessary. One may suggest ordering features with re-

spect to only their p-values, computed during the hypothesis testing step. It should be noted here that linear relationship between the values of Fisher's Score and p-values is not strong enough to lead exactly to the same feature orderings. Simulations performed by the authors demonstrated that for numeric values Fisher's Score based selection led to better orderings with respect to classifier accuracy. This fact justifies a two-step feature selection procedure for system calls. Regarding permissions, p-values and Gini Index based selection procedures did not lead to sufficient difference in detection accuracy. Nevertheless, a two-step selection procedure was used for the sake of method coherence.

In relation to classifier training, one has to choose desired number of predictors either on the basis of Fisher's Score values or Gini Index values. Note that there are no universal or generic valid thresholds for Fisher's Score and Gini Index values indicating suitability or unsuitability of a particular feature. Based on the outcomes of the feature selection process, we provided our expert judgement to determine the thresholds, selected the sets and verified their prediction performance by creating and testing the learning model.

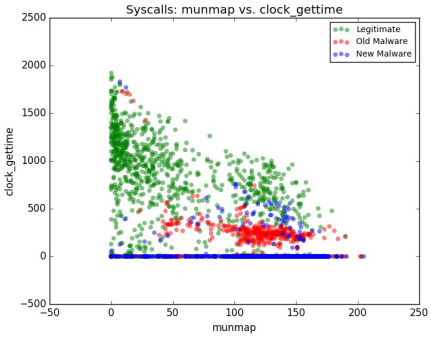


Figure 1: Scatter plot munmap vs clock_gettime.

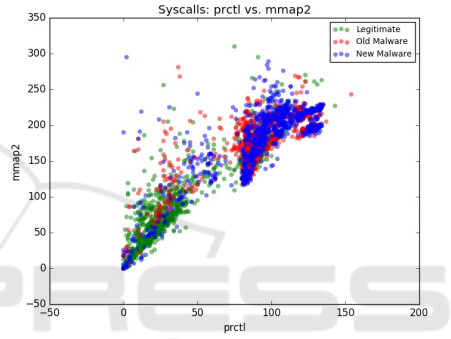


Figure 2: Scatter plot prctl vs mmap2.

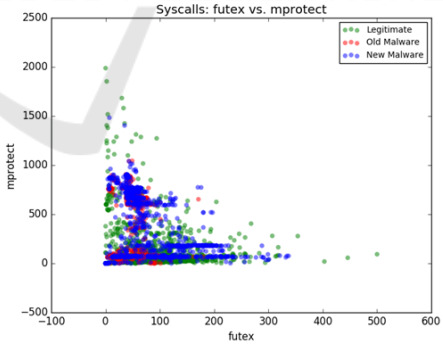


Figure 3: Scatter plot futex vs mprotect.

4 RESULTS & DISCUSSION

4.1 Results of Feature Selection and Ranking

We applied feature selection and classification methods to two different compound datasets: First one (namely L/O) includes 1000 legitimate and 1000 old malware samples, and second one (namely L/N) is composed by 1000 legitimate and 1000 new malware samples. Let us remind that each particular system call was treated as a numeric feature which results in 212 numeric features. Each particular permission was treated as a categorical feature (set or unset), which leads to 147 categorical features. Following the feature selection procedure described in Section 3.2, Welch's test demonstrated that for L/O dataset, 38 numeric features differed significantly between the legitimate and malicious applications for level of significance $\alpha = 0.05$, whereas this number was 43 for L/N dataset. In a similar manner, for the same level of significance, χ^2 filtered out 85 permissions for L/O dataset and 79 permissions for L/N dataset.

In the feature ranking step, Fisher's Score and Gini Index values were computed for numeric and categorical features respectively. This allowed ordering the features with respect to their discriminatory power. As mentioned before, there is no specific threshold on any of the methods performed to select or discard any particular feature, only data knowledge and expertise helps in this selection step. As all Fisher's Score (F) values were relatively low, we selected those system calls having $F > 0.15$. Regarding permissions, all Gini Index (G) values were relatively

Table 1: System Calls and Fisher's Score Values.

System Call	L/O	L/N
clock_gettime	0.84	1.11
munmap	0.75	0.57
readlinkat	0.69	0.59
connect	0.67	0.52
mmap2	0.63	0.47
prctl	0.61	0.53
madvise	0.54	0.48
ppoll	0.31	0.25
sigaction	0.29	0.30
sigaltstack	0.23	0.21
openat	0.22	0.16
mprotect	0.15<	0.19
futex	0.30	0.15<
rt_sigprocmask	0.24	0.15<
epoll_create1	0.23	0.15<
eventfd2	0.22	0.15<
getppid	0.22	0.15<
clone	0.21	0.15<
sendto	0.19	0.15<
recvfrom	0.18	0.15<
close	0.17	0.15<
getdents64	0.15	0.15<

Table 2: Permissions and Gini Index Values.

Permission	L/O	L/N
access_network_state	0.46	0.41
wake_lock	0.45	0.39
install_packages	0.42	0.41
read_phone_state	0.32	0.45
get_accounts	>0.47	0.47
system_alert_window	>0.47	0.46
get_tasks	>0.47	0.45
mount_unmount_file_systems	>0.47	0.44
vibrate	>0.47	0.44
access_fine_location	0.47	>0.47
bind_remoteviews	0.47	>0.47
use_fingerprint	0.47	>0.47
camera	0.47	>0.47
bluetooth	0.46	>0.47
read_logs	0.44	>0.47
send_sms	0.43	>0.47
read_contacts	0.43	>0.47
read_external_storage	0.33	>0.47

high so we selected those with $G < 0.47$. System calls possessing higher discriminatory power are listed, together with their Fisher's Score values, in Table 1. Similarly, Table 2 gives the selected permissions with their Gini Index values.

As a result of the second step, 21 features were selected for L/O dataset and 12 for L/N dataset among the system calls (11 of them were common in both datasets). All common system calls in L/N except `clock_gettime` have lower Fisher's Score values. Furthermore, there is only one additional discriminatory system call, `mprotect`, which has a relatively low score, that has been developed in the course of time (appears as potentially discriminatory feature in L/N dataset but not in L/O dataset). Based on that, it can be argued that separability between legitimate and new malware is less obvious, meaning that system call behaviour of malware has become more similar to legitimate as time has passed. Additionally, it can also be argued that beyond this separability fact, new malware has not developed a robust novel character.

Scatter plot graph given in Figure 1 shows an easily recognizable well-defined decision boundary that is formed by two of the most discriminatory system calls, `clock_gettime` and `munmap2`. As shown, old malware is gathered in a cluster which is located between legitimate and new malware regions. On the other side, decreased separability formed by system calls with relatively less Fisher's Score values, such as `prctl` and `mmap2`, is demonstrated in Figure 2. Although most of legitimate and new malware samples form their own clusters which can be separable from each other, boundaries are not so clear when compared to the graph given in Figure 1. Figure 3 shows the graph for two system calls having lower scores such as `futex` and `mprotect`. It is observed that despite some condensed regions occupied by one class, boundaries between old malware, new malware and legitimate apps mostly disappear.

According to Fisher's Score values, it can be derived that system calls that possess best discriminatory power are related to socket connection, process management or file operations. However, best predictor is the one which is related with clock time, showing the most different behaviour between malware and legitimate applications.

Based on Gini Index values (see Table 2) and the established threshold value, we identified that 13 permissions in L/O possess greater discriminatory power whereas 9 permissions have greater power in L/N (among the 147 permissions in total). New malware gained more separability from legitimate applications in features such as `wake_lock`, `access_network_state`, `install_packages`. They exceeded the threshold value in an additional five features which were below that value in old malware. On the other side, it has become closer to legitimate apps in 10 features (for instance, `read_phone_state`, `camera`, `send_sms`, or `read_contacts`). It can be argued that total discriminatory power of new malware has diminished to some

extent due to a reduction in the number of selected features, but in contrast to system calls, it gained new character.

Android OS has mainly three protection levels that determine policies for granting permissions to mobile apps: (1) Normal permissions which are automatically given to applications without explicit consent of the user, (2) Dangerous permissions that require explicit consent of the users to be granted, (3) Signature permissions which require that the app that uses the permission must have the same certificate as the app that defines the permission (Google, 2018). Features with greater discriminatory capabilities, which are identified by Gini Index in our study, do not belong to a single level. Among the 18 listed features in Table 2, only 7 of them belong to the dangerous level. This result indicates that malware and legitimate apps can also differ in permissions which do not seem risky.

It is important to note that, in our context, gaining character or having more discriminatory power means that the referenced dataset can better discriminate malware from legitimate apps by using the corresponding feature. It does not show that, for instance, malware uses that specific system call or permission more (or less) frequently than a legitimate app. However, as we utilized the same legitimate dataset, it is evident that the change in discrimination capabilities relies on the change of malware behaviour over time.

Table 3: Classification with System Calls.

# of features	L/O accuracy	L/N accuracy
Single Best Feature [1]	0.87	0.89
3 Best Common Features [2]	0.90	0.88
6 Best Common Features [3]	0.91	0.89
All 11 Common Features selected in both datasets [4]	0.93	0.89
All 22 Selected Features	0.97	0.91
All 212 Features	0.97	0.93

4.2 Verification of Selected Features with Classifiers

In order to verify the results obtained in Section 4.1, we built and tested classifiers with selected feature sets, grouping them in varied sizes. Recall that the filter methods that we use in this study treat each feature separately while measuring its discriminatory power, meaning that these sets do not guarantee higher accuracy due to, for instance, possible correlations among the selected features. This verification study is needed to show the validity of our findings.

We trained and tested *k*- Nearest Neighbours (kNN), Logistic Regression, Decision Tree, and Support Vector Machines (SVM) machine learning algorithms to the datasets. Among these methods, decision tree model demonstrated best accuracy results, therefore, this method was chosen for further analysis. Then decision tree model was applied to L/O and L/N datasets. As shown in Table 3, we computed accuracy value for different decision tree classifiers as a performance metric (i.e., accuracy is computed as the ratio of correctly classified samples to the total samples), using 5-fold cross-validation with varying feature set sizes for system calls. Corresponding confusion matrix of each classifier is given summarized in Table 4.

Table 4: Confusion Matrices for the Classification of System Calls.

# of features	Actual(L)/ Pred(L)	Actual(M)/ Pred(M)	Actual(L)/ Pred(M)	Actual(M)/ Pred(L)
Single Best L/O	265	265	29	41
3 Best L/O	261	279	31	29
6 Best L/O	293	259	25	23
11 Common L/O	299	262	24	15
22 Selected L/O	303	276	10	11
All (212) L/O	295	290	8	7
Single Best L/N	300	234	27	39
3 Best L/N	263	266	39	32
6 Best L/N	259	269	37	35
11 Common L/N	282	254	32	32
22 Selected L/N	272	268	36	24
All (212) L/N	279	281	19	21

Results of decision tree classifier model regarding system calls show that just a single feature, `clock_gettime` (highest Fisher's score value), was capable of discriminating malware from legitimate apps (in both L/O and L/N datasets) with an accuracy over 87 %. However, this feature provided better classification in L/N, which is in line with the higher Fisher's Score value of this feature in this dataset. In all other classifier models built, selected features provided better outcomes in L/O dataset, justifying that similarity of system calls behaviour between a legitimate app and malware is getting less obvious over time.

Accuracy results of classifiers increase as bigger feature set is covered in both datasets. Just the 22 selected features are enough to give the same accuracy performance than using all system calls (212) in L/O dataset. However, a similar point is not achieved in L/N dataset, indicating a decrease in the discrimina-

[1] clock_gettime

[2] clock_gettime, readlinkat, and munmap

[3] clock_gettime, readlinkat, munmap, connect, prctl and mmap2

[4] clock_gettime, readlinkat, munmap, connect, prctl, mmap2, madvise, ppoll, sigaction, sigaltstack, openat

tory power of the selected features. It can be derived from the confusion matrices given in Table 4 that classifiers are, in general, well-balanced in terms of false positive and false negative results, which are represented in the table as "Actual(L)/Predicted(M)" and "Actual(M)/Predicted(L)" respectively. Note that L refers to legitimate whereas M means malware. However, results of the best feature in L/O and L/N are slightly more skewed to false negatives whereas the classifiers with all 11 common features in L/O and all 22 selected features in L/N are more inclined to false positives.

Results regarding the application of decision tree classifier model to permissions are given in Table 5. Best feature provided accuracy values, 0.79 and 0.73, in L/O and L/N datasets respectively. These values are lower compared to the detection performance of best system call predictor. As shown, accuracy value in L/O was greater than in L/N. This fact was expected as the Gini Index score of the best feature in L/O dataset has a lower value than in L/N dataset, i.e. that it has more discriminatory power. Accuracy of the classifier that uses all selected features, in both datasets, reaches almost the same value obtained when all permissions are used, showing the effectiveness of feature selection in permissions.

Table 5: Classification with Permissions.

# of features	L/O accuracy	L/N accuracy
Single Best Feature [5]	0.79	0.73
4 Common Selected Features in both datasets [6]	0.86	0.85
All 18 Selected Features	0.94	0.92
All 147 features	0.95	0.92

Accuracy values of L/N were slightly lower than values of L/O when common or all selected permissions were used. This result suggests that as time has passed, separability between malware and legitimate applications has partly decreased regarding permissions.

Confusion matrices of classifiers built for permissions are summarized in Table 6. It can be extracted that most of classifiers are not well-balanced compared to the ones built on the basis of system calls. Results of the best and four common features in L/O are skewed to false negatives, but remaining ones are more balanced. L/N dataset provided unbalanced outcomes in each classifier. Best feature in L/N gave more false positives and remaining ones were inclined to false negatives.

Table 6: Confusion Matrices for the Classification of Permissions.

# of features	Actual(L)/ Pred(L)	Actual(M)/ Pred(M)	Actual(L)/ Pred(M)	Actual(M)/ Pred(L)
Single Best L/O	271	201	30	98
4 Common L/O	262	248	23	67
18 Selected L/O	284	280	19	17
All (147) L/O	281	290	14	15
Single Best L/N	186	253	117	44
4 Common L/N	281	227	29	63
18 Selected L/N	274	274	19	33
All (147) L/N	284	268	20	28

When outcomes of system calls and permissions are compared, it can be argued that their amount of loss regarding discriminatory power in L/N is different. All selected system calls in L/N gave an accuracy value of 0.91, showing a decline from 0.97 which was obtained in L/O. This value, 0.91, is below the accuracy result, 0.93, which was obtained in L/N when all system calls were used for the classification. On the other side, accuracy value declines from 0.94 to 0.92 for all selected permissions, which indicates a lower amount of loss than selected system calls. Accuracy value of 0.92, is equal to the result obtained by all permissions in L/N. Recall that, in Section 4.1, we identified a decrease from 21 to 12 in the number of system calls which exceeded the selection threshold in L/O and L/N datasets. Out of 12 system calls, just only two of them have higher Fisher's score in L/N. Contrarily, decline in permissions goes from 13 to 9, and more features, 5 of them, have higher discrimination capability in L/N. These findings support the results obtained in Section 4.1 so that system calls and permissions lost part of their discriminatory power in L/N, being the loss in system calls greater than the loss in permissions.

It is important to highlight here that our results regarding the change in selected feature sets indicate a concept drift. Comparison between system calls and permissions given above provides initial insights into the extent of this phenomenon. However, more complete derivations can be drawn with modelling the drift in the classifier. As we focus on feature selection and ranking in this paper, we postponed this modelling effort to our future work.

Table 7 demonstrates detection performance of a mixture of system calls and permissions (hybrid detection approach). Classifier was constructed using decision tree model within a 5-fold cross-validation setting. As can be seen, in both datasets, detection rates were higher compared to their previously built respective single type classifiers, using only static or only dynamic features.

[5] read_phone_state for L/O and wake_lock for L/N

[6] access_network_state, wake_lock, install_packages and read_phone_state for L/O and L/N

[7] clock_gettime and read_phone_state for L/O and

Table 7: Classification with System Calls and Permissions (Hybrid).

# of features	L/O accuracy	L/N accuracy
Best Two Features [7]	0.90	0.89
$4+11$ Common Selected Features in both datasets	0.95	0.92
$18+22$ Selected Features	0.97	0.94
All Features ($212+147$)	0.98	0.94

5 CONCLUSION & FUTURE WORK

Detection of mobile malware remains a significant challenge due to the rapidly evolving nature of the threat. Machine learning techniques have provided solutions to handle this problem. Although they have provided promising results, there is a room for improvement of the classifiers by the utilization of feature selection to obtain better classification accuracy, present the results in a more interpretable way and reduce required computational resources.

In this paper, we applied a feature selection and ranking procedure that consists of two consecutive steps, statistical hypothesis testing and filter feature selection method. The former enables us to select the features while the latter ranks them according to their discriminatory power. We used system calls and permissions as the feature categories due to their proven success in various research studies. Detection performance of selected features was evaluated in decision tree based classifiers. In order to analyze the impact of the changing behaviour on feature selection process, we induced classifiers with malware samples belonging to different time frames.

This study shows that a small number of selected features, such as 3-6 features, provide relatively high accuracy results. Even a single system call, the one possessing best Fisher's Score value in our feature domain, `clock_gettime`, provided accuracy values over 87%. We identified that 10-12% of the features are able to provide a discriminatory power which is very close to the power of using all features in both feature categories (system calls and permissions). Moreover, we identified that system calls and permissions of new malware samples are more similar to legitimate apps than the old ones. This result suggests a concept drift in these features. Additionally, feature rankings and classifier outputs indicate that system calls have lost more discriminatory power over time compared to permissions.

In this paper, we concentrated on feature selection and its implications on accuracy of machine learning classifiers. Findings regarding concept drift can be better explored and enhanced by precisely modelling this learning aspect in the classifier itself. Feature sets used in the classifiers could be enhanced by adding other static or dynamic categories. Also, required length of collection's time period for dynamic attributes such as system calls could be further investigated.

[7] clock_gettime and wake_lock for L/N

REFERENCES

Aggarwal, C. (2015). *Data Mining: The Textbook*. Springer International Publishing.

Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., and Rieck, K. (2014). Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings 2014 Network and Distributed System Security Symposium*, number February.

Cen, L., Gates, C. S., Si, L., and Li, N. (2015). A probabilistic discriminative model for android malware detection with decompiled source code. *IEEE Transactions on Dependable and Secure Computing*, 12(4):400–412.

Fedler, R., Schütte, J., and Kulicke, M. (2013). On the Effectiveness of Malware Protection on Android. Technical report, Fraunhofer, AISEC.

Feizollah, A., Anuar, N. B., Salleh, R., and Wahab, A. W. A. (2015). A review on feature selection in mobile malware detection. *Digital Investigation*, 13:22–37.

Google (2018). Permissions overview. Retrieved from: https://developer.android.com/guide/topics/permissions/overview.

Kim, H.-H. and Choi, M.-J. (2014). Linux kernel-based feature selection for android malware detection. In *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific*, pages 1–4. IEEE.

Lindorfer, M., Neugschwandtner, M., and Platzer, C. (2015). Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 422–433.

McAfee (2018). McAfee Mobile Threat Report Q1 2018. Retrieved from: https://www.mcafee.com/es/resources/reports/rp-mobile-threat-report-2018.pdf.

Nezhadkamali, M., Soltani, S., and Hosseini Seno, S. A. (2017). Android malware detection based on overlapping of static features. In *7th International Conference on Computer and Knowledge Engineering (ICCKE 2017), October 26-27 2017, Ferdowsi University of Mashhad*.

Qiao, M., Sung, A. H., and Liu, Q. (2016). Merging permission and api features for android malware detection. In *2016 5th IIAI International Congress on Ad-*

vanced Applied Informatics (IIAI-AAI), pages 566–571. IEEE.

Robert, C. (2014). *Machine learning, a probabilistic perspective*. Taylor & Francis.

Sahs, J. and Khan, L. (2012). A Machine Learning Approach to Android Malware Detection. In *2012 European Intelligence and Security Informatics Conference*, pages 141–147.

Statista (2018). Mobile os market share 2017. Retrieved from: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.

Unuchek, R. (2018). Mobile Malware Evolution 2017. Retrieved from: https://securelist.com/mobile-malware-review-2017/84139/.

Vidal, J. M., Orozco, A. L. S., and Villalba, L. J. G. (2017). Malware detection in mobile devices by analyzing sequences of system calls. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*, 11(5):606 – 610.

VirusTotal (2018). How to use VirusTotal Community - VirusTotal. Retrieved from: https://www.virustotal.com/es/documentation/virustotal-community/.

Welch, B. L. (1947). The generalization ofstudent's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35.

Yan, G., Brown, N., and Kong, D. (2013). Exploring discriminatory features for automated malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer.

Yuan, Z., Lu, Y., Wang, Z., and Xue, Y. (2014). Droid-Sec : Deep Learning in Android Malware Detection. In *Sigcomm 2014*, pages 371–372.

Appendix 2

Publication II

A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Differences in android behavior between real device and emulator: A malware detection perspective. In *Proceedings of the 6th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 399–404, 2019

2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)

Differences in Android Behavior Between Real Device and Emulator: A Malware Detection Perspective

Alejandro Guerra-Manzanares, Hayretdin Bahsi, Sven Nõmm
Department of Software Science
Tallinn University of Technology, Tallinn, Estonia
Email: {alejandro.guerra, hayretdin.bahsi, sven.nomm}@taltech.ee

***Abstract*— Behavioral data extracted from emulators or real devices, such as system calls, are utilized in research studies where machine learning models have been employed for mobile malware detection. However, these studies do not explore whether the selection of data source may have an impact on the performance of the models, assuming that both data sources generate similar outputs. We provide a comparative analysis of the data sets obtained from both sources by using statistical techniques, inducing learning models and demonstrating the impact of data source selection on detection models' performance. Our study shows that emulators generate more distinguishable data than real devices, meaning that designers of detection models should pay attention to the data sources utilized in the various steps of the machine learning workflow.**

***Index Terms*— mobile malware detection, machine learning, android malware, dynamic analysis, system call**

I. INTRODUCTION

Mobile malware is a major threat to mobile phone users. After the significant spike of mobile malware in 2018 [1], the threat is still posing a high risk to end users, evolving in complexity and scope [2], [3]. Malware authors focus their efforts on Android, the most popular mobile operating system, which dominates over 75% of the market share [3], [4]. Although recent versions of Android have improved its security capabilities, the majority of Android users still use old and not updated versions which increase even more their risk of being compromised [5]. In addition, the proved inefficient capabilities of malware detection software place Android users in a risky situation [6].

Aiming to overcome these limitations, machine learning (ML) models have been used to improve mobile malware detection capabilities with remarkable success [7]. The data needed by the ML algorithms are collected from Android applications in different ways, either running the application (dynamic analysis) or from the application *apk* file itself (static analysis) [8]. More specifically, static features, such as permissions or metadata, are extracted from *apk* files or source code directly, without actually running the application. Dynamic features, such as system calls and network flow, are collected when running application, thus they could be prone to change depending on the platform that executes the application (real or emulated device). Emulators are a cheap, easy to deploy and flexible approach to run applications, being the preferred approach in some of the research. The results based on emulators' tests are usually generalized to all Android devices (either real or emulated). Thus, if the behavior in emulators and real devices produce significant variations for the same *app*, using both data sources interchangeably with no caution could undermine the detection performance of the machine learning models.

In general, when dynamic analysis is compared to static analysis, the former is considered more reliable and less prone to be bypassed by malware [9] although both of them could be subject to various evasion and obfuscation techniques. Malware authors mostly target the manipulation of data extraction process to diminish the outcome of dynamic analysis whereas misguiding the analysis of extracted data is the main focus to defeat static analysis. In addition to the evasion techniques employed for data extraction, the behavioral differences obtained from emulators and real devices might be an additional factor that weakens the practical advantage of using dynamic features over static ones on ML models.

In this research, we analyzed the behavioral differences between Android emulation and real devices and their impact on ML based mobile malware detection models. We extracted the system calls triggered by malware and legitimate mobile app samples in emulator and real device and created a separate data set for each environment. We employed statistical methods to identify the features having higher discriminatory power and deduce the correlations between feature pairs in both data sets. Then, we created learning models that address to solve binary classification problems. Additionally, we conducted some experiments in a multi-class formulation in which device type and being malicious/legitimate constitute the different classes. This study is distinguished as, to the best of our knowledge, prior research has not provided a comprehensive analysis of such environments in the context of their impact on ML models.

This paper is structured as follows: Section II provides background information and a review of related literature while Section III describes the method followed in this study. Results of our experiments are presented in Section IV. Lastly, Section V concludes the study and states future work.

II. BACKGROUND AND LITERATURE REVIEW

The literature regarding behavioral-based malware detection is split into two in terms of data source: real device-based data, where the behavior of the application is monitored and logged using a real device, and emulator-based data, where the device is *virtualized* on a computer using a

978-1-7281-2949-5/19/$31.00 ©2019 IEEE

virtualization software environment, the emulator, that aims to reproduce the behavior of a real device.

Some of the studies preferred real devices as a data source. *Crowdroid* [10] is a behavior-based malware detection system in which a real device acquires the *app*'s system calls and sends them to a central server that runs a clustering algorithm to discriminate between benign and malware applications. MADAM [11] is a multi-level and behavior-based malware detection method in which the data collection was performed using a Samsung Galaxy Nexus running Android 4.3. In [12] 30 real devices with distinct versions of Android were used to perform malware detection. The authors of this study state that, in their context, using different versions of the OS, the nature of the Android kernel did not impact the performance of the detection method proposed. Other research conducted using real devices are [13], [14], [15], [16], [17], [18].

Droid-sec [19] is a malware detection system which combines static and dynamic features using deep learning techniques. Applications were run in *DroidBox* [20], a sandbox which emulates the Android environment and allows the dynamic analysis of Android applications. MALINE [21] is a dynamic Android malware detection technique that acquires behavioral data of applications, such as system calls, by running them on a customized build of the Android Software Development Kit, which includes Android Emulator [22], a virtual machine that emulates Android operating system. In [23], the authors used an Android 2.1 emulator to test their detection method stating that according their experience on the detection of some malware applications, the results using a real device and an emulator fully matched, thus the results on an emulator should be cogent. Other research studies using emulated devices are [24], [25], [26], [27], [28], [29].

As can be noticed, the referenced research on Android malware detection does not converge on the usage of any particular type of devices, either real or emulators, using a wide variety of approaches even in the studies belonging to each specific group. For example, among studies that use real devices, some conduct all the research on a single real device while others on different real devices (running different versions of Android). There is also no clear preference on the emulator platform used on the studies, ranging from the official Android emulator to customized Android malware detection sandboxes. To the best of our knowledge, the possible behavioral differences caused by the data collection environment have not been investigated. By not taking this fact into consideration, the generalization of the induced learning models' results appears to be questionable.

III. METHOD

In this research, ML based malware detection is performed on the basis of system calls, issued by malware and legitimate applications. Network traffic is another behavioral component but it describes the behavior of the application partially and is out of scope of the present paper. For each dataset, we conducted the analysis including the following items:

- Descriptive statistics
- Correlations between features
- Feature selection (identifying the discriminatory power of the features)
- Accuracy of ML classifier models in which training and testing are conducted with the same data set (named as single data set approach)
- Accuracy of ML classifier models in which testing is conducted with another data set (named as cross-data set approach)

The data set used in this research is composed of 220 Android applications, distributed evenly as follows:

- 110 benign applications collected randomly from *APKMirror* repository between 2017 and 2018. Tested to be malware-free using *VirusTotal* detection engine.
- 110 randomly selected malware applications from VirusTotal academic malware data set, belonging to the same period of time as the legitimate applications.

Data set samples were installed and executed during 1 minute in a real device and an emulator. The details of the real device and emulator device are described as follows:

- Real device $\rightarrow$ Samsung Galaxy A6 with Android 8.0.
- Emulated device $\rightarrow$ Samsung Galaxy S8 with Android 8.0, using *GenyMotion* emulation environment.

Android Debug Bridge (ADB) was used to install, execute, monitor, log and uninstall each application. The system calls issued by the main process of each application during the running time were logged and collected using *strace*. No interaction was performed with the applications aside from booting them up using *monkey* tool for Android. On both devices the same basic Android 8.0 configuration was setup: security measures were disabled, Wi-Fi connection enabled, SD card, Google Play installed and no SIM card.

The following subsections give the details of each analysis item.

A. Data Preparation and Descriptive Statistics

212 system calls are defined in the standard C library for Linux kernel, called Bionic library, used in Android OS. For each application, a frequency vector that reflected the ratio of each system call issued was formed using the application's logged data.

Descriptive statistics were calculated for the whole acquired data set, dividing them according to application type (benign or malware) and collection source (real or emulated). In this step, raw data were used, so all the syscalls collected were processed, even those not defined in the Bionic library. Table I shows the measures of central tendency calculated: mean ($\bar{\chi}$) and median ($\widetilde{x}$), and the measures of dispersion: range, standard deviation (s) and inter-quartile range (IQR) for both kind of system calls, the ones included in the Bionic library (referenced with subscript b, e.g., $\bar{\chi}_b$) and the ones not included (referenced with subscript nb, e.g., $\bar{\chi}_{nb}$).

We identified that emulator data included just 95 system calls out of the 212 syscalls defined in the Bionic library, meaning that the remaining system calls were not issued. On the other side, we captured only 71 different system calls from the real device acquired data. As can be observed on

TABLE I
DESCRIPTIVE STATISTICS ON ACQUIRED DATA

	Real Device		Emulator	
	Benign	Malware	Benign	Malware
$\bar{x}_b$	12993	11610	12601	12010
s_b	30357	29496	15899	19853
range	[36, 289715]	[368, 265746]	[1121, 106076]	[28, 101867]
$\tilde{x}_b$	6213	4033	7561	4343
IQR_b	9245	5300	9632	6174
$\bar{x}_{nb}$	1469	1367	0	1
s_{nb}	3641	3575	1	2
range	[7, 34281]	[24, 33032]	[0, 11]	[0, 11]
$\tilde{x}_{nb}$	618	410	0	0
IQR_{nb}	951	532	0	0

Table I, the descriptive statistics show differences on most of the statistics computed among the devices. More specifically, applications show more dispersion of the data in the real device than in the emulated device, as can be confirmed by the wider range and greater standard deviation on real device data. Furthermore, in the emulated device, almost all the system calls issued are found in the Bionic library while in the real device there are much more system calls that do not belong to this specific library. These facts suggest that the behavior of the same applications on real and emulated devices is different.

After this step, the data was processed, thus being normalized for further use in the machine learning models. As the amount of system calls issued by each application could differ for the same collection time (1 minute), the acquired data was normalized using the maximum number of system calls issued by the specific application, thus each attribute value reflects the proportion of each feature among the total amount of system calls issued by the application (logically, the sum of all proportions is equal to 1, reflecting the total amount of system calls issued by the application). It is worth mentioning that he collection time, 1 minute, was established based on our experiments as the most relevant to understand the discriminatory power of the features, thus selected as the optimal discriminatory time-frame. Due to the space limitation, further information is not provided in this paper.

As the outcome of this step, two data sets were created: real data set and emulator data set. Each data set contains exactly the same features (212) and exactly the same number of instances (220): the same 110 benign and the same 110 malware applications. The differences rely on the features' values obtained from real device or emulator for each specific application.

B. Feature Correlation

Pearson's correlation coefficient (ρ) was applied to the normalized data sets. Pearson's ρ is a statistical criterion that quantifies the linear relationship between pairs of variables, ranging from [-1, 1]. Extreme values show perfect correlations (-1 for negative and +1 for positive) while a value of 0 means that no linear correlation exists between the assessed variables. In order to analyze the correlation of system calls for each of the data sets, Pearson's ρ was computed for all the 212 with the rest of variables in a pair-wise fashion. Figure 1 shows the features that at least had one strong correlation with another feature, either positive or negative, $|\rho| >= 0.80$, in the real device data set while Figure 2 provides the same information for the emulator data. Graphically, the darker the color, the greater the correlation.

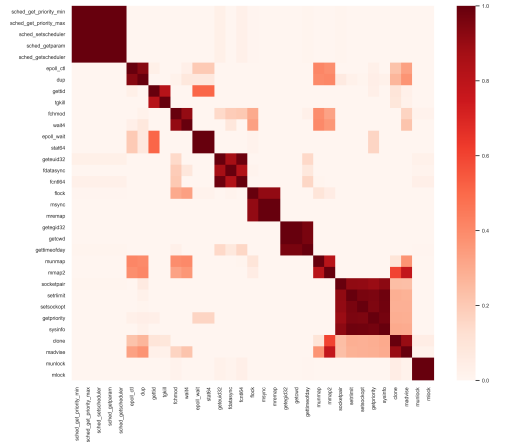


Fig. 1. Strongly correlated system calls from real device data

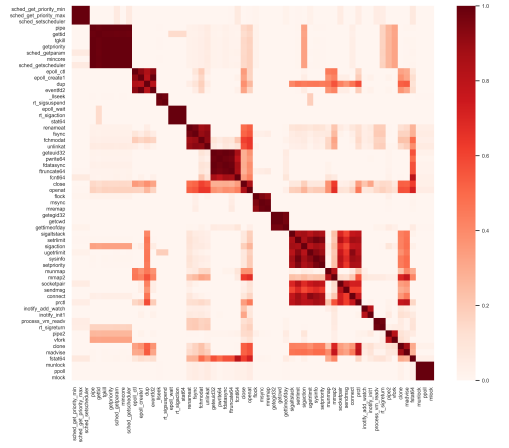


Fig. 2. Strongly correlated system calls from emulator device data

As can be observed on Figure 1, 33 features were found to have at least one strong correlation with another system calls while this value reached 60 features in the case of the emulated device, as shown in Figure 2. 30 features were common in both feature sets. As can be observed when comparing both figures, more features are correlated in the case of emulated data, suggesting that the behavior in the emulator tends to group features together, while in the real device this grouping is less frequent. Recall that emulator and real device used 95 and 71 different system calls respectively, thus it can be deduced that emulated device tends to use more system calls and in a grouped fashion (60 out of 95 features showed at least one strong correlation) while the real device

uses less system calls and in a less correlated manner (33 out of 71 features showed at least one strong correlation).

C. Feature Selection

Feature selection methods, which allow data dimensionality reduction, help to reduce the risk of overfitting and improve model's performance while reducing training time. In this study, after the data were normalized, Fisher's score was used to select most discriminatory features. Fisher's score is a filter feature selection technique that allows to rank features according to their discriminatory power among the different data classes (in this case, malware or benign). More specifically, Fisher's score is a statistical criterion that quantifies the ratio of the average inter-class separation to the average intra-class separation, as defined by the formula:

$$F = \frac{\sum_{j=1}^{k} p_j(\mu_j - \mu)^2}{\sum_{j=1}^{k} p_j \sigma_j^2}$$

where μ refers to the global mean of the data for the individual feature; μ_j and σ_j, stand for the mean and standard deviation of the data instances belonging to j class, and p_j is the fraction of instances that belong to j class. Based on that, greater values of Fisher's score suggest greater discriminatory power of the feature among the classes.

TABLE II
FISHER'S SCORE RANKING ON BOTH DATA SETS

Real		Emulator	
System call	F	System call	F
setsockopt	0.21	clock_gettime	0.92
socketpair	0.20	setrlimit	0.23
sysinfo	0.20	connect	0.21
setrlimit	0.20	sysinfo	0.20
getpriority	0.17	ugetrlimit	0.18
sendmsg	0.13	sigaction	0.18
prctl	0.09	setpriority	0.16
clock_gettime	0.07	getuid32	0.14
sched_yield	0.04	readlinkat	0.14
read	0.04	socketpair	0.13

Fisher's score ranking of the 10 best features for each data set and their scores (F) are provided in Table II. As can be observed in Table II, only 4 system calls are common on both rankings (highlighted in grey). Considering these best features, the minimum Fisher's score value in the emulator dataset is 0.13 which is considerably greater than the minimum score in the real device data set, 0.04. Among the common ones, *clock_gettime* is so distinguished that it has a very high discriminatory power in emulator data set (0.92) but the value is considerably low in the real data set. The closest feature to *clock_gettime*, *setrlimit*, has a score, 0.23, which is quite low. To sum up these findings, the top 10 Fisher's score values on real device are lower than the values obtained for the emulator, suggesting that the emulator data could be more distinguishable among classes.

D. Classifier models: single and cross data set approaches

Malware detection could be considered as a binary classification problem that can be tackled using the machine learning approach to discriminate between malware and benign applications. In this research, we trained and evaluated five different traditional supervised machine learning models used for binary classification problems: $k-$Nearest Neighbors (k-NN), Logistic Regression (LR), Support Vector Machines (SVM), Decision Tree (DT) and Random Forest (RF) algorithms.

Binary classification models were trained and validated using two distinct approaches:

- Single data set approach: the same data set was used for training and validation. Different sizes of feature sets were used to build the models. Models were cross-validated using 5-fold cross-validation.
- Cross data set approach: the models were trained with one data set and validated using the other data set. Different combinations of features were used to build the models.

After that, the best overall classifier algorithm of the previous task was used to handle a multi-class classification problem (i.e. four-class classification). In this regard, the two data sets were merged, the labels were changed to distinguish between application type coming from different sources (real or emulated). A four-class classification problem was built in order to discriminate the four possible options: real device malware, real device benign, emulated device malware and emulated device real application. The purpose of this experimental setup was to test whether a machine learning model could be able to discriminate between malware from benign while also identifying the data source. If so, it could be argued that the behavior of the same application on each device is different enough to allow the machine learning algorithm to discriminate them as if they really were two different applications. If that were the case, such learning models could be utilized in the situations where the data source is known at the training stage but unknown at testing stage, which might help to avoid misclassifications due to the fact of mixing data from different sources.

The performance metrics of all models built and tested are reported. They are described as follows:

- Accuracy: ratio of correctly classified test instances among all test instances.
- Precision: fraction of positive (malware) instances correctly classified among all the positive classified instances.
- Recall: fraction of positive (malware) instances correctly classified among all the real positive instances.

IV. RESULTS AND DISCUSSION

A. Binary classification

Machine learning classification models were built and 5-fold cross-validated. The library used to build all the models was Python's *scikit_learn* library. As the purpose of this research was not to reach an optimal classifier but to highlight any differences between different real and emulated Android devices, the hyper-parameters used for all the classifiers were the default values provided by the library.

k-Nearest Neighbors, Support Vector Machines, Logistic Regression, Decision Tree and Random Forest models were trained and tested with same source data (either real or

emulator) and different features (either all features or best 1, 5, 10 features according each data set Fisher's score ranking). Accuracy, precision and recall performance metrics for each model are provided in Table III. As can be observed in

TABLE III

PERFORMANCE METRICS OF SINGLE SOURCE ML MODELS

k-Nearest Neighbors

	Real				Emulator			
	All	1	5	10	All	1	5	10
Accuracy	0.7727	0.5272	0.5136	0.7545	0.8727	0.8090	0.8090	0.8272
Precision	0.7807	0.5158	0.5075	0.7599	0.8654	0.8258	0.8258	0.8385
Recall	0.7636	0.9272	0.8818	0.7454	0.8909	0.8	0.8	0.8363

Support Vector Machines

	Real				Emulator			
	All	1	5	10	All	1	5	10
Accuracy	0.6227	0.6000	0.6181	0.6	0.8227	0.7863	0.7818	0.8045
Precision	0.6823	1.0	1.0	0.7104	0.9044	0.9178	0.9125	0.9185
Recall	0.3909	0.2000	0.2363	0.3636	0.7181	0.6272	0.6181	0.6636

Logistic Regression

	Real				Emulator			
	All	1	5	10	All	1	5	10
Accuracy	0.7090	0.6681	0.6590	0.6590	0.8227	0.8318	0.8318	0.8363
Precision	0.7386	0.7654	0.7394	0.6725	0.9044	0.9113	0.9113	0.9119
Recall	0.6636	0.4818	0.4909	0.6272	0.7181	0.7363	0.7363	0.7454

Decision Tree

	Real				Emulator			
	All	1	5	10	All	1	5	10
Accuracy	0.7818	0.6636	0.6545	0.7545	0.8227	0.7909	0.7727	0.8227
Precision	0.8063	0.8580	0.8239	0.7539	0.8390	0.8005	0.7679	0.8189
Recall	0.7636	0.3909	0.3909	0.7545	0.8	0.8	0.8	0.8454

Random Forest

	Real				Emulator			
	All	1	5	10	All	1	5	10
Accuracy	0.8272	0.6636	0.6545	0.7590	0.8727	0.7863	0.7954	0.8454
Precision	0.8420	0.8580	0.8391	0.8053	0.9358	0.8138	0.7981	0.8616
Recall	0.8	0.3818	0.3909	0.7	0.8	0.7636	0.7999	0.8363

Table III, the models built and tested based on emulator data show better overall performances than the ones built and tested using real device data. This fact goes in line with the greater Fisher's score values obtained in the previous steps, suggesting that separability of data points among labels on emulator is greater than in real device. Random Forest algorithm outperformed the other classifiers in most of the experiments, be it real device or emulator. Based on that, Random Forest was used to build cross-source data set models. These models are featured to be models which are built (trained) using data from one data set and are tested with data belonging to the other data set. The number of estimators for the Random Forest algorithm was established to 100 (i.e., *scikit_learn*'s default value). Different combinations of the best features were selected to build the models, belonging to one data set or the other, as it is specified in Table IV where, for the sake of easy comparison, only the accuracy of each model is reported.

According to Table IV, when the model is trained with real data and using at least 10 features, the detection of emulator data is better than detection of real data, which should be explained by the presence of *clock_gettime* feature in the feature set, the highest ranked feature in the emulator data set. In any other case, training with real data, the models show better performance detecting real data. When the model is trained with emulator data, this pattern is also present and it can be observed that, in general, emulator data is better detected than real data using any feature from the feature set selected by Fisher's score criterion for this data set, even with just 1 feature the model achieves 78.63% accuracy. Based on that, it can be argued that, for the same samples, emulator based acquired data provide better detection patterns and capabilities than real device acquired data.

TABLE IV

ACCURACY OF CROSS-DATA SET CLASSIFICATION MODELS

			Test Data	
		Features	Real	Emulator
Train Data	Real	All	0.8272	0.8795
		Best Real 1	0.6636	0.5045
		Best Real 5	0.6545	0.6340
		Best Real 10	0.7590	0.8659
		Best Emu 1	0.6045	0.4249
		Best Emu 5	0.6954	0.6409
		Best Emu 10	0.7924	0.8181
	Emulator	All	0.7931	0.8727
		Best Real 1	0.6045	0.4568
		Best Real 5	0.6909	0.6727
		Best Real 10	0.7613	0.8636
		Best Emu 1	0.6022	0.7863
		Best Emu 5	0.6613	0.7954
		Best Emu 10	0.6636	0.8454

B. Multi-class classification

The differences highlighted in the previous paragraphs suggested that it might be possible to train a classifier able to discriminate between the four possible labels: legitimate/malware from real device and legitimate/malware from emulator. In order to build that model, the two binary class data sets were merged to build a single four-class Random Forest classifier. Feature selection was performed as in the previous steps for the merged data set, results are shown in Table V. The classifier performance summary using different number of features and 5-fold cross-validations is provided in Table VI. Macro-averaged metrics are retrieved as the test data distribution is not imbalanced towards any of the classes. Figure 3 shows an example of a confusion matrix extracted from the tested model when all features are used on training and testing stages. As can be seen in Table V, the most discriminatory system call among these four labels (*openat*) was not found in none of the individuals data sets rankings, a fact that emphasizes the difference between each of the four labels. According to Table VI, a model using all features is capable of discriminate accurately 86% of the data points, while with just 10 features, 81.59% . The confusion matrix in Figure 3 confirms that the model, even not optimized (using default parameters), can successfully discriminate between data point sources showing more accurately results for the emulator data than the real device data. This fact emphasizes the possibility of building a classifier model able to predict the type and source of an application.

TABLE V

TOP 10 - FISHER'S SCORE RANKING OF MERGED DATA SET

System call	F
openat	0.95
clock_gettime	0.76
readlinkat	0.56
fstatat64	0.49
epoll_pwait	0.49
setsockopt	0.42
ugetrlimit	0.41
faccessat	0.34
mkdirat	0.25
socketpair	0.24

V. CONCLUSIONS AND FUTURE WORK

Machine learning based models which are developed for mobile malware detection use either emulator or real device as a data source when they address the issue from a dynamic analysis perspective. Emulators have been more

TABLE VI
PERFORMANCE OF MULTI-CLASS RANDOM FOREST MODEL

	Features				
	All	1	3	5	10
Accuracy	0.8613	0.5272	0.7409	0.7568	0.8159
Precision	0.8645	0.4073	0.7452	0.7630	0.8241
Recall	0.8613	0.5272	0.7409	0.7568	0.8159

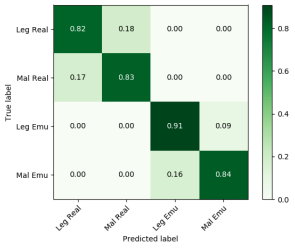


Fig. 3. Normalized confusion matrix

favored during the data collection process as they can be readily scaled to run various detection tasks by using less computational resources. However, this choice assumes that emulators and real devices are indistinguishable in terms of produced data and thus model performance. Our experiments suggest that there is an important difference in the behavioral aspect of applications running in real or emulated devices. These differences are emphasized by training a classifier that allows to discriminate among the same data points collected from different sources. These differences can hinder the capabilities of a classifier trained on one source of data to detect data coming from another source. In this regard, we have shown that a four-class classification model can be constructed to establish the source of the data with notable accuracy. The causation behind the differences of behavior evidenced by this study remains unclear and will be addressed in our future work.

REFERENCES

[1] V. Chebyshev, "Mobile malware evolution 2018." [Online]. Available: https://securelist.com/mobile-malware-evolution-2018/89689/

[2] McAfee, "McAfee Mobile Threat Report." [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf

[3] V. Chebyshev, F. Sinitsyn, D. Parinov, B. Larin, O. Kupreev, and E. Lopatin, "IT threat evolution Q1 2019. Statistics." [Online]. Available: https://securelist.com/it-threat-evolution-q1-2019-statistics/90916/

[4] Statista, "Mobile operating systems' market share worldwide from January 2012 to December 2018." [Online]. Available: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[5] Android, "Distribution dashboard." [Online]. Available: https://developer.android.com/about/dashboards

[6] R. Fedler, J. Schütte, and M. Kulicke, "On the Effectiveness of Malware Protection on Android," AISEC, Tech. Rep., 2013.

[7] B. Baskaran and A. L. Ralescu, "A study of android malware detection techniques and machine learning," in *MAICS*, 2016.

[8] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab, "A review on feature selection in mobile malware detection," *Digital investigation*, vol. 13, pp. 22–37, 2015.

[9] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 2007, pp. 421–430.

[10] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[11] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, 2016.

[12] J. M. Vidal, A. L. S. Orozco, and L. J. G. Villalba, "Malware detection in mobile devices by analyzing sequences of system calls," *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 11, no. 5, pp. 594–598, 2017.

[13] X. Xiao, X. Xiao, Y. Jiang, X. Liu, and R. Ye, "Identifying android malware with system call co-occurrence matrices," *Transactions on Emerging Telecommunications Technologies*, vol. 27, no. 5, pp. 675–684, 2016.

[14] V. Wahanggara and Y. Prayudi, "Malware detection through call system on android smartphone using vector machine method," in *2015 Fourth International Conference on Cyber Security, Cyber Warfare, and Digital Forensic (CyberSec)*. IEEE, 2015, pp. 62–67.

[15] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "andromaly: a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, pp. 161–190, 2012.

[16] C. Da, Z. Hongmei, and Z. Xiangli, "Detection of android malware security on system calls," in *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. IEEE, 2016, pp. 974–978.

[17] G. Canfora, E. Medvet, F. Mercaldo, and C. A. Visaggio, "Detecting android malware using sequences of system calls," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*. ACM, 2015, pp. 13–20.

[18] X. Xiao, P. Fu, X. Xiao, Y. Jiang, Q. Li, and R. Lu, "Two effective methods to detect mobile malware," in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, vol. 1. IEEE, 2015, pp. 1041–1045.

[19] Z. Yuan, Y. Lu, Z. Wang, and Y. Xue, "Droid-sec: deep learning in android malware detection," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 371–372.

[20] DroidBox, "DroidBox." [Online]. Available: https://github.com/pjlantz/droidbox

[21] M. Dimjašević, S. Atzeni, I. Ugrina, and Z. Rakamaric, "Evaluation of android malware detection based on system calls," in *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*. ACM, 2016, pp. 1–8.

[22] Google, "Run apps on the Android Emulator." [Online]. Available: https://developer.android.com/studio/run/emulator

[23] Y.-D. Lin, Y.-C. Lai, C.-H. Chen, and H.-C. Tsai, "Identifying android malicious repackaged applications by thread-grained system call sequences," *computers & security*, vol. 39, pp. 340–350, 2013.

[24] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, "Identifying android malware using dynamically obtained features," *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 1, pp. 9–17, 2015.

[25] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs," in *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*. IEEE, 2016, pp. 104–111.

[26] L. Singh and M. Hofmann, "Dynamic behavior analysis of android applications for malware detection," in *2017 International Conference on Intelligent Communication and Computational Techniques (ICCT)*. IEEE, 2017, pp. 1–7.

[27] A. Ferrante, E. Medvet, F. Mercaldo, J. Milosevic, and C. A. Visaggio, "Spotting the malicious moment: Characterizing malware behavior using dynamic features," in *11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2016, pp. 372–381.

[28] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2. IEEE, 2015, pp. 422–433.

[29] A. Guerra-Manzanares, S. Nõmm, and H. Bahsi, "In-depth feature selection and ranking for automated detection of mobile malware," in *2019 5th International Conference on Information Systems Security and Privacy*, 2019, pp. 274–283.

Appendix 3

Publication III

A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. Time-frame analysis of system calls behavior in machine learning-based mobile malware detection. In *International Conference on Cyber Security for Emerging Technologies (CSET)*, pages 1–8, 2019

Time-frame Analysis of System Calls Behavior in Machine Learning-Based Mobile Malware Detection

Alejandro Guerra-Manzanares, Sven Nõmm, Hayretdin Bahsi
Department of Software Science, TalTech
Tallinn, Estonia
{alejandro.guerra, sven.nomm, hayretdin.bahsi}@taltech.ee

***Abstract*—Dynamic features are frequently used in the machine learning based approaches to detect malicious applications on Android devices. These features are constructed by collecting the system calls observed during a certain period of time. In spite of the popularity of this approach, very little attention has been paid to the analysis of the length of the collection time-frame and its impact on the detection performance of induced learning models, which constitutes the scope of this research. Such analysis helps to understand the accuracy and performance trade-off in data collection efforts taking place at the various stages of the machine learning workflow. Our time-frame analysis also addresses different data collection environments, emulator and real device, and the variations in detection capabilities in the case of detecting recent or older malware. System calls of 330 benign and malicious applications, collected on different time periods, are monitored and logged for each minute-long interval for a total of fifteen minutes. First, distribution of the system calls is analysed. After, the discriminatory power of each system call is evaluated cumulatively for each minute-long interval. Fisher's score is used to assess the discriminatory power of each feature. It is revealed that the system calls observed during the first minute possess the highest discriminatory power, whereas the discriminatory power of the system calls observed on greater time-frames is lower. Finally, this finding is tested by training and evaluating traditional machine learning classifiers.**

***Index Terms*—system calls, mobile malware, machine learning, dynamic behavior, malware detection, time analysis**

I. Introduction

Mobile malware poses a real threat to mobile devices' users, which has evolved in scope and complexity of the attacks [1] [2]. Cyber attackers have directed the majority of their efforts to compromise Android OS, the widely-used open-source mobile operating system [3], locating its users as the main targets of their attacks [4].

Machine learning algorithms have been tested, using several approaches, to overcome the main weaknesses and limitations of signature-based antivirus detection in the mobile malware landscape [5] with remarkable success [6]. In this regard, static and dynamic features of Android applications have been used to create learning models to detect mobile malware with promising results [7]. Static features are extracted directly from the application's source code without needing to execute it while dynamic features are acquired when the application is running on an Android device, either real or emulated [8]. Whereas static features (e.g. permissions or API calls) are usually collected in its whole at once and rarely change on different collections from the same application, dynamic features (e.g. system calls or network traffic) are prone to changes and increase in volume depending on other variables such as the collection time and user-interaction on different collections on the same application. As machine learning algorithms heavily rely on data quantity and data quality, the time variable appears to be of remarkable importance in the case of the usage of dynamic features, as it may affect to quality of the data when increasing data quantity (e.g. adding noise or irrelevant data) which may directly cause a remarkable impact on machine learning algorithm's detection ratio.

The main objective of this research is to elucidate the impact of the collection time on data quality, and more specifically, whether it exists a specific time-frame that may be capable of providing optimal accuracy performance when using system calls data to induce learning models (i.e., whether more system calls data encompassing longer time-frame provide better detection ratio). We also analyse the optimal data collection time-frame for detecting malware belonging to different years. As Android is the most widely used mobile operating system, we address malware detection in that environment.

In order to achieve our research purpose, first, we analyse the distribution of system calls per application in each minute. Then, we conduct a discriminatory power analysis of the data collected in different time-frames by using Fisher's score. Finally, we induce learning models to evaluate the accuracy performance with varying feature sets.

Our contribution is unique as our time-frame analysis is more comprehensive in terms of covering different data collection environments (i.e., emulation and real device) and testing with new and old malware.

This paper is organized as follows: Section II provides a literature review and background information while Section III explains the methodology followed in this research. Section IV shows the experimental results obtained and Section V concludes the study and states the future work.

II. Background information & Literature review

Machine learning based malware detection using dynamic features requires the need of running the applications for a certain amount of time and/or user-interaction in order to acquire the data. Generally, longer data collection times provide more data, which may usually be related to better outcomes when using machine learning models. However,

the amount of time can also provide more noisy data thus lowering the data quality that could harm the performance of the learning models. The existent research literature does not show any particular trend or common practice regarding the collection time, although system calls are the most widely-used feature when dynamic features are used to induce learning models [7].

Some studies use pseudo-random or human user-interaction as a method for the collection of the application behavior data, being the Monkey [9] the most used tool to generate the pseudo-random user-interactions or events. The Monkey uses by default no delay between the events, generating them as rapidly as possible, thus there is no guarantee about that the data obtained can belong to the same time-frame, unless the same seed is used for each application to generate the same pseudo-random chain of events, like in [10] where the tests included 1, 500, 1000, 2000 and 5000 pseudo-random injected events. In [11], [12] and [13], 1000 pseudo-random events were generated to obtain the behavior of the application, 500 pseudo-random events in [14] while in [15], two different settings were used to generate and extract system call data: 2500 pseudo-random interactions and 5 minutes human interactions. In [16], human user-interactions were used, ranging from 6 to 60 interactions to detect both self-written malware and real malware. A different approach was used in [17], where a custom behavioral testing tool, called *Component Traversal*, was used to execute all the activities and services defined by each application. The average execution time was 41 seconds. There is also some of the literature using this approach does not provide any information about the number of events injected like [18], [19] or the number of human events performed [20].

The time comparison issues that may arise from the uncontrolled pseudo-random generated events can be overcome with the usage of time-limited collections. In this regard, in [21], benign and malware mobile games system call data was collected for 30 and 60 seconds. The shorter time-frame did not provide any discriminatory differences between applications while the longer provided discriminatory patterns among classes that could be used to detect and classify mobile gaming applications. Different data sets were run for 10 seconds per application in [22] and 20 seconds in [23]. In [24], 44 applications were used by a human for 10 min each whereas in [25] the Monkey was used to mimic human interaction in a 10 min limited collection. Also some studies using this approach do not provide any information about the acquisition time like [26] and [27].

Finally, a third approach of the research dealt directly with the acquisition of a specific number of system calls or sequences. In this regard, [28] evaluated the impact of different lengths of system calls on application's boot up (i.e., 500, 1000, 2000, 2500 system call sequences) while [29] used a fixed amount of system calls per application (i.e, first 2000 system calls) to perform malware detection on different malware data sets.

III. Methodology

A. Data set

The data set used in the experimental set-up is composed of 330 Android x86 applications distributed as follows:

- 110 random benign applications collected from *APKMirror* repository between 2017 and 2018. Checked as malware-free using *VirusTotal* malware detection engine. Named indistinctly as "legitimate dataset" or "benign dataset" in this research.
- 110 randomly chosen malware applications from VirusTotal academic malware data set [30], belonging to the time frame between 2017 and 2018. Named as "new malware" in this research.
- 110 randomly selected malware applications from the Drebin malware dataset [31], dating from 2010 to 2012. Named as "old malware" in this research.

As can be noticed, two distinct malware data sets were selected from different time-frames but only one benign data set was selected with more recent applications. The rationale behind this selection is that in order to analyse the time-evolution we have to analyze all kinds of applications we can encounter *in the wild* which encompasses old and new (recent) malware and just recent applications. This is coherent with the mobile malware detection practices as mobile phones typically suffer from back-compatibility issues, so that older legitimate applications do not usually work with the recent OS, due to changes in application requirements and constant changes in hardware and software from new OS developments. Nevertheless, malware detection systems usually include signatures of all malware samples, including old and new ones.

B. Data acquisition

This research focuses on the extraction and usage of dynamic features, i.e. system calls, to perform mobile malware detection on the Android OS environment. Over 200 distinct system calls are defined in the Bionic x86 library, the standard C library for Android, which are monitored, extracted and logged for this research.

All the applications are installed, executed, monitored, logged and uninstalled on Android devices, both in a real device (Samsung Galaxy A6) and an emulated device (Samsung Galaxy S8 emulated using *GenyMotion* emulation software). The rationale behind using two different types of devices is that the existing literature use them interchangeably, using emulators as a cheap and scalable approach to real devices, thus preferred in some researches [32] and taken into account in this experimental set-up. Android 8.0, the most deployed version of the Android OS [33], is used as the operating system running on both devices, with identical configuration. Each application is executed and allowed to run without any user interaction for 15 minutes. Application's behavioral data, i.e. system calls issued by the application's main process, are logged using *strace* tool. Consequently, fixed-collection time with no user-interaction is analyzed in this experimental setup while any kind of user-interaction, either real or emulated, is

out of the scope of this present research and will be part of the future work.

The outcome of this step is a log file for each application and for each device (real and emulated), which contains all the system calls issued by each specific application for the whole acquisition time, established in 15 minutes.

C. Data processing

Applications' logged data is analysed from a time-frame evolution perspective. This is performed in a two-fold approach: time-specific system call frequency analysis and time-cumulative discriminatory power perspective. They are described as follows:

1) Time-specific system call frequency analysis: In this step, the number of system calls issued by each application for each single running minute is analyzed using histograms. So, as the outcome of this step, a histogram is obtained for each application logged data, providing an overview of the absolute frequency (i.e. count) of the system calls issued by the application over time, on a minute-long basis analysis.

2) Time-cumulative discriminatory power feature analysis: In this step, Fisher's score (F), which is a statistical criterion that allows to assess the discriminatory power of numeric features among the different classes (i.e. malware or legitimate application), is used to evaluate the discriminatory power evolution of each feature (i.e. each system call) over time among the classes, in a time-cumulative basis. More specifically, Fisher's score quantifies the ratio of the average inter-class separation to the average intra-class separation, providing a measure of the discriminatory power of the feature. Greater magnitudes relate with greater discriminatory powers. Fisher's score is calculated as follows for each feature:

$$F = \frac{\sum_{j=1}^{k} p_j (\mu_j - \mu)^2}{\sum_{j=1}^{k} p_j \sigma_j^2}$$

where μ refers to the global mean of the data on the particular feature; μ_j and σ_j, relate to the mean and standard deviation of the data points belonging to class j for the specific feature respectively, and p_j refers to the proportion of data points belonging to class j. So, in this research, as the outcome of this step, for each feature (system call) we obtain a line-graph showing the discriminatory power evolution for each minute which allows us to assess the discriminatory power evolution of each feature over time.

D. Machine Learning models validation

From the data processed on the previous step, machine learning binary classification models are built and validated. As the main purpose of the machine learning models induced is to evaluate empirically the previous findings and the overall performance and trends of the classifiers, we use widely used traditional machine learning algorithms for classification issues to induce the malware classification models. In relation to that, we do not perform any hyper-parameter optimization of the machine learning algorithms, keeping the default configurations that Python's $scikit\_learn$ library provides. The models built aim to provide empirical support to the results obtained on previous steps, thus providing a general overview of the classifiers' performance. In this regard, there is room for improvement, so that the results can be enhanced by optimizing the models' hyper-parameters, but is out of the scope of this present research.

Three widely used machine learning algorithms for classification problems are evaluated: Random Forest, $k-$Nearest Neighbors and Support Vector Machines. All models are validated using 5-fold cross-validation, which aims to provide a better estimation of the predictive model's performance against $unseen$ data (i.e., data that has not been used to build the model) than the regular fixed train-test split when the data set used is small. The performance metric reported is accuracy, a comprehensive metric that stands for the ratio of correctly classified test instances among all the test instances. Accuracy range varies from 0 to 1. Greater accuracy score imply greater classification performance.

IV. Results

A. Time-specific system call frequency analysis

The absolute frequency of system calls is used in this step to construct application-wise histograms, as shown as an example in Figure 1. For the sake of interpretability, histogram bars are slightly spaced resembling a bar graph. The horizontal axis is split and numbered from 1 to 15, each corresponding to a minute fraction while the vertical axis accounts for the absolute frequency of system calls issued for the application in each time slot. Figure 1 shows an example of an application that provided data for each minute of the total collection time. More concretely, it corresponds to a *new malware* data set sample (i.e., *apk*'s package name: *air.com.bitrhymes.bingo*) that issued approximately 10000 system calls in the first minute and a relatively constant amount of slightly less than 4000 each subsequent minute until the end of the data acquisition process. As can be observed, the application issued system calls even when no interaction was performed along the 15 minutes run-time.

Contrarily, histograms in Figure 2 and 3 are provided as examples of applications that did not provide data for each minute of the whole 15-minute run-time. More specifically, Figure 2 shows an example of an *old malware* application (i.e., *apk*'s package name: *anohito.ha.ima*) that issued system calls only for the first minute, over 7500 system calls, while Figure 3 provides an example of a *legitimate* application (i.e., *apk*'s package name: *com.ms.office365admin*) which issued a vast amount of system calls only for the first three minutes with an irregular pattern, issuing more system calls on the second minute than in the first and stopping on the third. It can be argued that these applications stopped issuing system calls before the ending of the acquisition time because either it finished its boot-up before the run-time timeout established on 15 minutes, reaching an idle state, or was blocked waiting

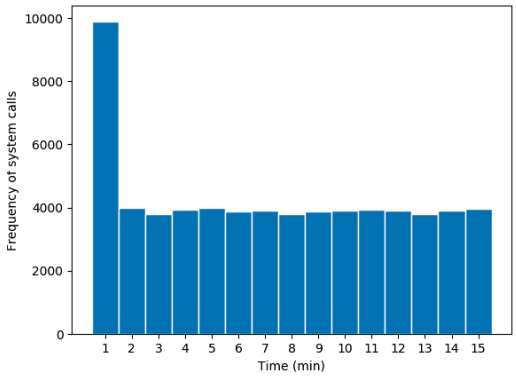


Fig. 1. System call histogram of a *new malware* application sample

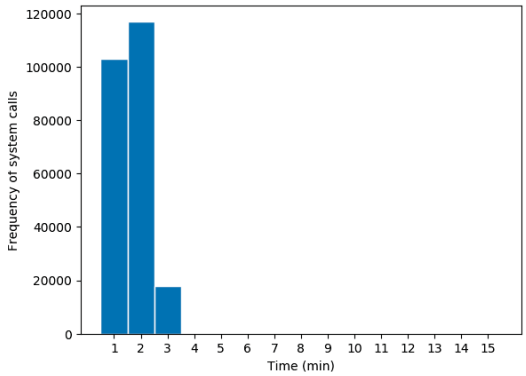


Fig. 3. System call histogram of a *legitimate* application sample

for some input, expecting some user-interaction or data that never happened. Examples on Figure 1, Figure 2 and Figure 3 are provided as an example of histograms obtained from logged data analysis, but they are not representative of any class. Such an examples of histograms can be found on all the data sets, either malware or benign.

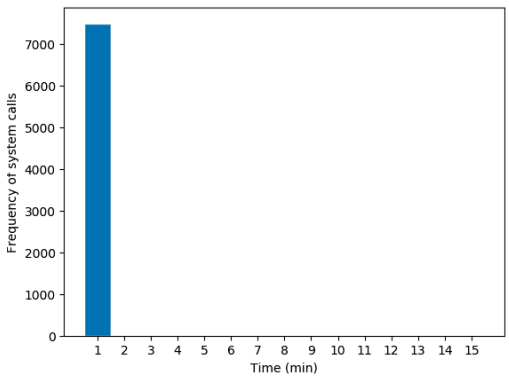


Fig. 2. System call histogram of an *old malware* application sample

A summary of the results obtained is provided in Table I and Table II, which provide the frequency distribution of applications (i.e., relative frequency) in relation to the maximum time they provided data on run-time, both on an emulator and real device. Relative frequency provides information about the occurrence of an outcome (e.g., providing data only for one minute) among all the possible outcomes (i.e., the 15 time-frames splits on a minute basis). Table I provides an overview and grouped data summary extracted from Figure 4 while Table II provides the same information from Figure 5. In both Figure 4 and Figure 5, the top graph (green data) shows information about legitimate applications (referenced as L in the corresponding tables), the center graph (blue data) shows information about *old malware* applications (referenced as O in the tables) while the bottom graph (red data) shows information about *new malware* applications (referenced as N in the tables) . As can be observed, there is a great number of applications that provided data only for one minute and got stuck while many others provided information for the whole acquisition time. Those time-frames are the ones with consistently higher frequency along all data sets, even when they only encompass one minute data and not four minute data like the other groups. The time-frame of 6-10 minutes is consistently less frequent. These general patterns and facts are found on all classes of applications and not specifically linked to malware or legitimate applications.

Data distribution was also analysed regarding whether the application issued the maximum number of system calls in the first minute split (called *1 min spike* in this research and referenced as such in the corresponding tables) or in any of the subsequent time-slots. In this regard, Table I and Table II reference on their last row the relative frequency of applications that show the *1 min spike* on each data set. As can be seen, on all data sets the vast majority of applications (at least 88% in the worse case) issue the maximum number of system calls in the first minute, issuing less in the subsequent minutes, or nothing at all. So, based on the aforementioned, the first minute of application's boot-up appears to be the most productive in terms of system calls issuing, consistently, among all the different data sets and especially when a real device is used.

TABLE I
SYSTEM CALLS STATISTICS ON EMULATOR

Observed Fact	Frequency distribution		
	Leg	Old	New
1 min data	0.2631	0.2536	0.4545
2 to 5 min data	0.1929	0.1594	0.1188
6 to 10 min data	0.0614	0.0869	0.0769
11 to 14 min data	0.0350	0.1594	0.0209
15 min data	0.4474	0.3405	0.3286
1st min spike	0.9035	0.8985	0.8811

TABLE II
SYSTEM CALLS STATISTICS ON REAL DEVICE

Observed Fact	Frequency distribution		
	Leg	Old	New
1 min data	0.4166	0.3550	0.4685
2 to 5 min data	0.3166	0.1739	0.1678
6 to 10 min data	0.0166	0.0507	0.0349
11 to 14 min data	0.0083	0.0217	0.0349
15 min data	0.2416	0.3913	0.2937
1st min spike	1.0	0.9855	0.9370

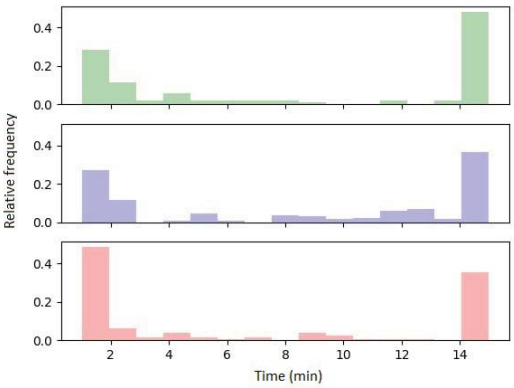


Fig. 4. Emulator data set frequency distribution

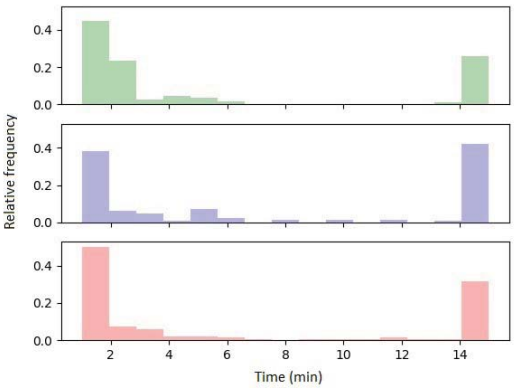


Fig. 5. Real device data set frequency distribution

B. Time-cumulative discriminatory power feature analysis

In this step, the single benign data set is merged with the two malware data sets, providing two different mixed benign-malware data sets. These new mixed data sets are named L/O (when benign data is mixed with old malware data set) and L/N (when benign data is mixed with new malware data) for each of the devices used (real or emulated). For each possible combination device and data set (i.e. L/O data on real device, L/N on real device, L/O on emulated device and L/N on emulated device) Fisher's score is calculated in a cumulative manner for each minute (e.g., minute 3 data contains all the data up to that minute).

Fisher's score is used in machine learning applications as a feature selection method as it measures the separability of the data among classes, allowing to select the best features to train classifiers. In this regard, the greater the value, the greater the separability of the data, implying that the data is less mixed and more clustered among labels.

As a result of this step, line graphs are obtained for each minute on each of the four possible combinations, showing the evolution of data separability regarding each possible system call over time, as shown in the example of Figure 6. The horizontal axis is split and numbered from 1 to 15, each corresponding to a minute fraction while the vertical axis accounts for the Fisher's score value of the specific feature in each time-slot. In this graph, dotted lines represent the evolution of values using L/N data set, thus indicating in the legend with the letter *R* the results corresponding to Real device and with the letter *E* to the emulated device. Solid lines represent the evolution of values using L/O data set, indicating also the device source, whether R or E. As a result, line graphs are obtained for each of the system calls showing the trend or evolution over time in each of the four cases.

Figure 6, shows the line graph of the feature (i.e. *clock_gettime*) with the greatest Fisher's score value, obtained using L/O data set run on emulator. In this case, Fisher's score value is relatively stable along all the running period. In the other three cases, the greatest Fisher's score value is reached in the first minute and lowering as time passes. This latter trend is confirmed on the line graphs shown in Figure 7, which provides the time evolution of the features that achieved the greatest Fisher's scores among all the evaluations. Only 19 features from over 200 defined in the *Bionic x86* library provided a Fisher's score value over 0.10 in at least one of the possible data set/device cases.

As can be spotted in Figure 7, the maximum value of Fisher's score on all features and in almost all cases is achieved in the first minute, diminishing its initial value over time. This fact suggests that the separability of data is greater in the first minute, being the best source of data to perform classification and malware detection. As time passes, the data is becoming more mixed, thus less suitable to perform proper classification. It can also be noticed from Figure 7 that data belonging to emulators are, in general across the features, more clearly separable than data obtained from real devices, providing different results using the same data set, as suggested in [32]. As can be noticed on Figure 6 and Figure 7, L/O emulator data is the case where there exist more prevalence of the diminishing discriminatory trend on features among all the possible cases.

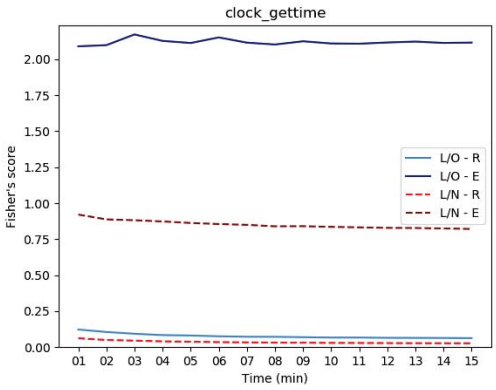


Fig. 6. *clock_gettime* syscall Fisher's score time evolution

C. Machine Learning models validation

Machine learning models are induced and 5-fold cross validated. Random Forest, Support Vector Machines (SVM) and k-Nearest Neighbors algorithms are tested and evaluated, providing similar trends and results. For the sake of brevity, only the model that provided the best accuracy performance, outperforming the other two, is reported. In this regard, Random Forest algorithm, which is a decision tree-based ensemble method, provides the best performance and its results are provided in Figure 8. The hyper-parameters used are the default ones used by the library implementation, so that the number of estimators parameter was set to 100.

The three graphs presented in Figure 8, show the classifier's performance on the four possible cases and its accuracy evolution using data from different time frames. More specifically, the left-most graph on Figure 8 shows the models built with data from the 18 features provided in Figure 7. In this graph, it can be easily observed the decrease in accuracy for L/O data (both in emulator and real device) and a more irregular pattern with lower general accuracy value on the models built with L/N data. The center graph in Figure 8 provides the results when the models are induced using the previous 18 features plus *clock_gettime*, the one that provided the greatest overall Fisher's score. It can be observed that the same decrease pattern appears as in the previous graph when using Real L/O data and Emulator L/O stays more stable while the overall accuracy on emulator and Real L/N data is lower. This fact confirms the pattern highlighted in Figure 6, where emulator data on both data sets provided great Fisher's score values, thus related to an increased data separability and improved accuracy value of the classifier. On the other side, the impact of *clock_gettime* in accuracy increase is slightly better in each L/N data. The right-most graph on Figure 8 shows the models induced using all the features (over 200), with similar trends and slightly improved accuracy values. However, the figures between all features and 19 features are so close that the latter

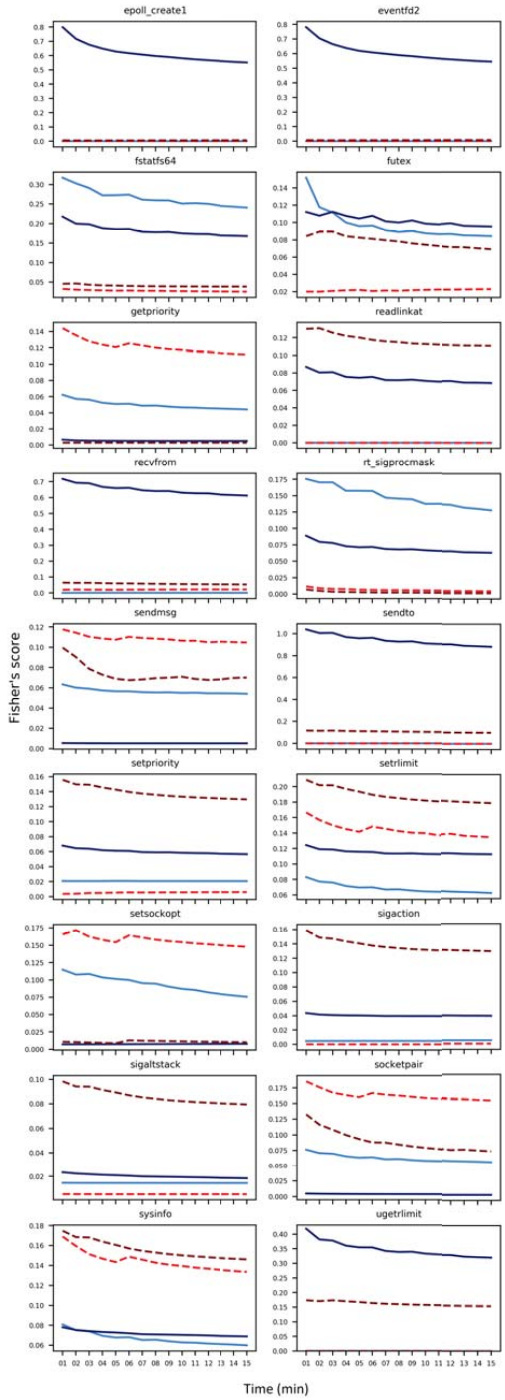


Fig. 7. Best Fisher's score syscalls time evolution

can be a viable option considering the shortcomings of using all features in consuming computational resources.

As can be stated from the observation of the graphs provided in Figure 8, in general, the accuracy obtained using one-minute data provides better or similar accuracy values when compared to including more data in a longer time duration. In this regard, we can observe that in some cases (specially when using old malware), there is a decline in the accuracy performance over time. In other cases, there is no prominent increase or decrease in the accuracy metric. Based on the aforementioned observations, we conclude that, in this case, covering longer time interval than 1 minute does not constitute a more convenient option as it does not significantly improve the classifier's performance, when using accuracy as performance metric. However, it is important to note that 1 minute is definitely a better option for the data including old malware when feature selection is applied.

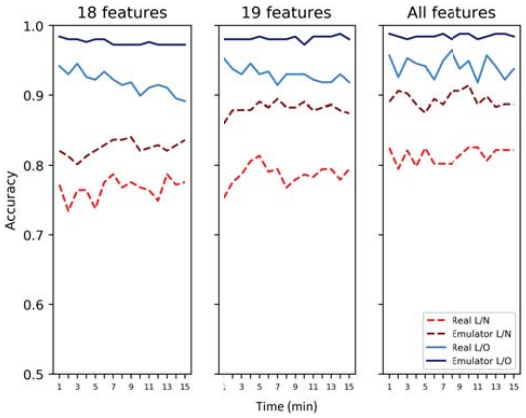


Fig. 8. Random Forest models' accuracy

D. Threats to validity

The results of this study highlight differences in the discriminatory power of features along time. The main weaknesses for the generalization of these findings are the limited data set size (110 benign, 110 *old* malware and 110 *new* malware samples) and the limited number of emulator/real devices tested (two devices, one emulator and one real device). In this regard, a bigger data set and the additional testing in more emulators/real devices could help to overcome this limitations, in order elucidate whether these differences are only between the chosen pair of emulator/real device or a more general issue that should be take into consideration in the behavioral machine learning-based mobile malware detection. These threats will be tackled deeply within our future work.

V. Conclusions & Future Work

System calls are one of the most widely used features when dealing with dynamic analysis in machine learning based mobile malware detection. As the acquisition of a dynamic features, including system calls, require the execution the application, different approaches have been used to collect them, mainly differing in the collection time and the usage of human or software-based user-interaction. This research focused on the analysis of the impact of the collection time on the separability of the data in a learning model and its time-based evolution, when no user interaction is performed. Applications were just executed and let run freely for the whole acquisition time (15 minutes).

The usage of Fisher's score in conjunction with frequency distribution analysis has demonstrated that most applications perform in the first minute the maximum number of system calls (the so-called *data spike* in this research) and has demonstrated that, specially in some cases (e.g., when emulator is used and L/O data set), short-time data collection (i.e. 1 minute) may provide greater data separability thus, consequently, leading to greater accuracy performance metrics on mobile malware classifiers than long-time data collection (e.g. 15 minutes). Machine learning classifiers may accurately be optimized for such purposes with this input data achieving great accuracy performances. In this regard, the collection of more data, during more time, may not lead to provide better data. Thus, in this case, data quality, in the form of greater separability, might be jeopardized by noisy data when the collection lasts for a longer time.

The underlying causation of the existing differences highlighted in this research between the data sets obtained from the emulator and the real device remains unclear. In this regard, the potential impact of the different environment variables, such as kernel version and network connection type, will be further investigated in order to obtain a deeper understanding of the possible sources that may explain the behavioral deviations found in this study. The explanation and investigation of these behavioral divergences in addition to the increase of the data set used and their test with additional emulators and real devices will constitute part of our future work. Finally, as already stated in Section III.B, the influence of user-interaction will also be explored in later stages of our future work, thus providing a complementary perspective to the findings of this present research, where no user-interaction was performed during the experimental set-up.

References

[1] V. Chebyshev, "Mobile malware evolution 2018." [Online]. Available: https://securelist.com/mobile-malware-evolution-2018/89689/

[2] R. Samani and G. Davis, "McAfee Mobile Threat Report Q1, 2019," 2019. [Online]. Available: https://www.mcafee.com/enterprise/en-us/assets/reports/rp-mobile-threat-report-2019.pdf

[3] A. Holst, "Global mobile OS market share in sales to end users from 1st quarter 2009 to 2nd quarter 2018," 2019. [Online]. Available: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[4] P. Security, "Android devices 50 times more infected with malware compared to iOS," 2019. [Online]. Available: https://www.pandasecurity.com/mediacenter/mobile-security/android-more-infected-than-ios/

[5] R. Fedler, J. Schütte, and M. Kulicke, "On the effectiveness of malware protection on android," *Fraunhofer AISEC*, vol. 45, 2013.

[6] D. Geneiatakis, G. Baldini, I. N. Fovino, and I. Vakalis, "Towards a mobile malware detection framework with the support of machine learning," in *International ISCIS Security Workshop*. Springer, 2018, pp. 119–129.

[7] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab, "A review on feature selection in mobile malware detection," *Digital investigation*, vol. 13, pp. 22–37, 2015.

[8] A. Kapratwar, F. Di Troia, and M. Stamp, "Static and dynamic analysis of android malware." in *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*. SciTePress, 2017, pp. 653–662.

[9] Android, "UI/Application Exerciser Monkey." [Online]. Available: https://developer.android.com/studio/test/monkey

[10] M. Dimjašević, S. Atzeni, I. Ugrina, and Z. Rakamaric, "Evaluation of android malware detection based on system calls," in *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*. ACM, 2016, pp. 1–8.

[11] S. Zhang and X. Xiao, "Cscdroid: Accurately detect android malware via contribution-level-based system call categorization," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 193–200.

[12] X. Xiao, P. Fu, X. Xiao, Y. Jiang, Q. Li, and R. Lu, "Two effective methods to detect mobile malware," in *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, vol. 1. IEEE, 2015, pp. 1041–1045.

[13] X. Xiao, X. Xiao, Y. Jiang, X. Liu, and R. Ye, "Identifying android malware with system call co-occurrence matrices," *Transactions on Emerging Telecommunications Technologies*, vol. 27, no. 5, pp. 675–684, 2016.

[14] T. Bhatia and R. Kaushal, "Malware detection in android based on dynamic analysis," in *International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*. IEEE, 2017, pp. 1–6.

[15] P. Vinod, A. Zemmari, and M. Conti, "A machine learning based approach to detect malicious android apps using discriminant system calls," *Future Generation Computer Systems*, vol. 94, pp. 333–350, 2019.

[16] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[17] S. Hou, A. Saas, L. Chen, and Y. Ye, "Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs," in *2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW)*. IEEE, 2016, pp. 104–111.

[18] L. Singh and M. Hofmann, "Dynamic behavior analysis of android applications for malware detection," in *2017 International Conference on Intelligent Communication and Computational Techniques (ICCT)*. IEEE, 2017, pp. 1–7.

[19] H. Alptekin, C. Yildizli, E. Savas, and A. Levi, "Trapdroid: Bare-metal android malware behavior analysis framework," in *2019 21st International Conference on Advanced Communication Technology (ICACT)*. IEEE, 2019, pp. 664–671.

[20] F. Tchakounté and P. Dayang, "System calls analysis of malwares on android," *International Journal of Science and Technology*, vol. 2, no. 9, pp. 669–674, 2013.

[21] M. Jaiswal, Y. Malik, and F. Jaafar, "Android gaming malware detection using system call analysis," in *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*. IEEE, 2018, pp. 1–5.

[22] A. Ahsan-Ul-Haque, M. S. Hossain, and M. Atiquzzaman, "Sequencing system calls for effective malware detection in android," in *IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2018, pp. 1–7.

[23] M. R. Amin, M. Zaman, M. S. Hossain, and M. Atiquzzaman, "Behavioral malware detection approaches for android," in *IEEE International Conference on Communications (ICC)*. IEEE, 2016, pp. 1–6.

[24] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""andromaly": a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, pp. 161–190, 2012.

[25] A. Ferrante, E. Medvet, F. Mercaldo, J. Milosevic, and C. A. Visaggio, "Spotting the malicious moment: Characterizing malware behavior using dynamic features," in *2016 11th International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2016, pp. 372–381.

[26] G. Canfora, F. Mercaldo, and C. A. Visaggio, "A classifier of malicious android applications," in *2013 International Conference on Availability, Reliability and Security*. IEEE, 2013, pp. 607–614.

[27] W. Yu, H. Zhang, L. Ge, and R. Hardy, "On behavior-based detection of malware on android platform," in *2013 IEEE global communications conference (GLOBECOM)*. IEEE, 2013, pp. 814–819.

[28] J. M. Vidal, A. L. S. Orozco, and L. J. G. Villalba, "Malware detection in mobile devices by analyzing sequences of system calls," *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 11, no. 5, pp. 594–598, 2017.

[29] A. Guerra-Manzanares, S. Nõmm, and H. Bahsi, "In-depth feature selection and ranking for automated detection of mobile malware," in *Proceedings of the 5th International Conference on Information Systems Security and Privacy*, vol. 1. SciTePress, 2019, pp. 274–283.

[30] VirusTotal, "How to use VirusTotal Community - VirusTotal," 2018. [Online]. Available: https://www.virustotal.com/es/documentation/virustotal-community/

[31] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and Explainable Detection of Android Malware in Your Pocket," in *Proceedings 2014 Network and Distributed System Security Symposium*, 2014.

[32] A. Guerra-Manzanares, H. Bahsi, and S. Nõmm, "Differences in android behaviour between real device and emulator: A malware detection perspective," in *2019 6th Int Conference on Internet of Things, Systems, Management & Security (IOTSMS)*. IEEE, 2019, forthcoming.

[33] Google, "Distribution dashboard," accessed on: Oct. 5, 2019. [Online]. Available: https://developer.android.com/about/dashboards/

Appendix 4

Publication IV
A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Kronodroid: Time-based hybrid-featured dataset for effective android malware detection and characterization. *Computers & Security*, 110:102399, 2021

COMPUTERS & SECURITY 110 (2021) 102399

Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

KronoDroid: Time-based Hybrid-featured Dataset for Effective Android Malware Detection and Characterization

*Alejandro Guerra-Manzanares**, *Hayretdin Bahsi*, *Sven Nõmm*

Department of Software Science, Tallinn University of Technology

ARTICLE INFO

Article history:
Received 2 March 2021
Revised 1 July 2021
Accepted 4 July 2021
Available online 9 July 2021

Keywords:
Android malware
Dataset
Mobile malware
Malware detection
Malware analysis

ABSTRACT

Android malware evolution has been neglected by the available data sets, thus providing a static snapshot of a non-stationary phenomenon. The impact of the time variable has not had the deserved attention by the Android malware research, omitting its degenerative impact on the performance of machine learning-based classifiers (i.e., concept drift). Besides, the sources of dynamic data and their particularities have been overlooked (i.e., real devices and emulators). Critical factors to take into account when aiming to build more effective, robust, and long-lasting Android malware detection systems. In this research, different sources of benign and malware data are merged, generating a data set encompassing a larger time frame and 489 static and dynamic features are collected. The particularities of the source of the dynamic features (i.e., system calls) are attended using an emulator and a real device, thus generating two equally featured sub-datasets. The main outcome of this research is a novel, labeled, and hybrid-featured Android dataset that provides timestamps for each data sample, covering all years of Android history, from 2008-2020, and considering the distinct dynamic data sources. The emulator data set is composed of 28,745 malicious apps from 209 malware families and 35,246 benign samples. The real device data set contains 41,382 malware, belonging to 240 malware families, and 36,755 benign apps. Made publicly available as *KronoDroid*, in a structured format, it is the largest hybrid-featured Android dataset and the only one providing timestamped data, considering dynamic sources' particularities and including samples from over 209 Android malware families.

1. Introduction

Android operating system (OS) has become the reference OS in the mobile platform ecosystem. Empowered by Google, the free and open-source OS has consistently dominated the mobile operating system market since 2012. From 2017, it has been shipped with over 70% of smartphones, showing no signs of a decrease in the near future (Statista, 2021a). Over 99% of the mobile OS market share is covered when iOS devices are added, placing Apple's proprietary OS as the major alternative to Android OS (Statista, 2021a). The global dominance of Android OS and the wealth of data stored by smartphones make Android users an attractive target for cyber-attackers. After a massive outbreak in 2016, Android malware attacks have plateaued but still remain a constant and evolving threat for the end-users (Chebyshev, 2019). New and more sophisticated malicious applications are found on a daily basis, evidencing the constant evolution of the phenomenon both in new malware trends (e.g., ransomware) and sophistication

* Corresponding author.
E-mail address: alejandro.guerra@taltech.ee
(A. Guerra-Manzanares).
https://doi.org/10.1016/j.cose.2021.102399

(Microsoft 2020). In 2020, over 480,000 new Android malware samples were discovered monthly (Statista 2021b). However, these figures only reflect the detected malware, which could be considered as just the *tip of the iceberg*. Malware authors are always ahead in innovation, achieving sophistication levels that might let malware hide effectively, acting undetected (Broersma, 2020).

But the situation might be even worse, as Android malware can be and has been found anywhere, even in *Google Play*, affecting millions of users (McGowan, 2020). Despite the remarkable efforts of Google and Android phone vendors to implement security mechanisms at software (e.g., Bouncer (Oberheide and Miller, 2012) and Google Play Protect (Google 2021)) and hardware levels (e.g., Samsung Knox (Samsung, 2021)), malware authors have always found the way to circumvent them (Cimpanu, 2019; Lakshmanan, 2020). Furthermore, traditional antivirus techniques (i.e., signature/fingerprint) have been proved ineffective to detect zero-day or *unknown* malware in Android devices (Fedler et al., 2013; Withwam, 2020).

However, the war is not yet lost. The application of machine learning techniques to Android malware detection has shown outstanding results, using a wide variety of app features (Feizollah et al., 2015), even with zero-day, repackaged, and obfuscated malware (Grace et al., 2012), overcoming the limitations of traditional detection methods (Faruki et al., 2013). Machine learning (ML) uses data properties or characteristics (i.e., features) to build effective systems that find statistical patterns to solve the problem at hand (e.g., malware detection). The performance of a machine learning system is strictly related to data quantity, but more significantly to data quality (Cortes et al., 1994; Sessions and Valtorta, 2006). Therefore, machine learning-based Android malware detection systems largely depend on the quality of the data features to perform effectively and accurately. The Android malware data feed into the machine learning algorithms are a critical element for the overall success, reliability, and generalization capabilities of the ML-based malware detection system.

1.1. *Datasets for Android Malware Detection*

The phenomenon of malware detection in Android systems has been investigated since the early years of the popular OS, but with increased attention after the wide adoption of mobile networks, the ubiquity of smartphones, and the rise of mobile applications (*apps*). Roughly 3 million apps are available just in *Google Play* at the present time (Google 2021), added and removed daily. With billions of downloads per year (Iqbal, 2020), apps are the main attack vector to perpetrate attacks in Android devices and the critical component to build effective malware detection systems. Despite this fact and the availability of a few newer Android data sets in the recent years, *old* malware data sets have monopolized the research. These datasets have been widely used as reference data to build machine learning systems, even in recent studies. A fact that neglects malware evolution and its change over time and poses at severe doubt the generalization capabilities, reliability, and effectiveness of the models induced using *old* datasets to detect novel malware.

The most *popular* datasets for Android malware research are summarized in Table 1. For each dataset reported, the following information is provided.

- **Name** - denomination or acronym that uniquely identifies the dataset.
- **Composition** - two non-negative integers separated by a slash symbol ("/"). The first value indicates the number of malware apps included in the data set and the second value the quantity of benign/legitimate apps if any.
- **Time-frame** - reported time period where the apps composing the dataset were collected.
- **Access.** - short for accessibility. It indicates whether the data set is publicly available. A $\surd$ indicates the dataset is available while X reports when it is not (i.e., project discontinued).
- **APK** - indicates whether the data set provides the executable files (i.e., *.apk* files). A $\surd$ indicates the dataset provides the *apk*s while a X reports the negative case.
- **Features** - indicates whether the data set provides collected features from the apps, in a structured format. A X reports no features are provided. When features are provided the type is specified using a keyword: *static*, *dynamic*, or *hybrid*. More about this terminology in Section 2.2.
- **Year** - the year of publication of the data set's research article.
- **Ref.** - short for references. Composed of a pair of references inside brackets. The first specifies the research article where the data set was presented while the second refers to the data set's website resource.
- **Citat.** - short for citations. Provides the number of citations of the dataset's reference article according to *Google Scholar* at the time of writing. It gives a rough measure of the usage of the data set for research.

As shown in Table 1, the vast majority of the research on Android malware has focused on *MalGenome* and *Drebin* datasets. When added together, they cover 91% of the total citations for the datasets included in Table 1 (i.e., 3,761 out of 4,134). A notable fact considering that most of the datasets were not published recently. However, the most recent samples in these two popular datasets were collected in 2012, almost a decade ago. A significant amount of time when technological advances are considered, making them *outdated* or *old* in terms of malware evolution. Furthermore, their size is relatively small, especially in the case of *MalGenome*, composed of just 1,260 samples from 49 malware families and no legitimate samples. *Drebin*, collected in a similar time-frame, provides a more complete malware data set, composed of 5,560 samples belonging to 179 malware families. Nevertheless, Drebin may contain duplicated data (Irolla and Dey, 2018), which makes the *usable* malware data set smaller and does not provide access to the legitimate *apk*s. Despite all these facts, both datasets have been widely used, even in recent studies, as the main source of malware data to build machine learning-based malware detection systems that aim to detect recent Android malware (Wu et al., 2020; El Fiky, 2020).

The third most referenced dataset, the *Android Malware Dataset* (AMD), is a larger and more recent dataset that spans a wider time-frame in the Android history but accounts for a

Table 1 – Most *popular* Android malware datasets.

Name	Composition	Time-frame	Access.	APK	Features	Year	Ref.	Citat.
MalGenome	1,260 / 0	2010-2011	X	√	X	2013	(Zhou and Jiang, 2012, Zhou and Jiang, 2015)	2282
Drebin	5,560 / 123,453	2010-2012	√	√	static	2014	(Arp et al., 2014, Braunschweig, 2020)	1479
CICAndroidBot	1,929 / 0	2010-2014	√	√	X	2015	(Kadir et al., 2015, U. of New Brunswick 2020)	58
Kharon	20 / 0	2011-2015	√	√	static	2016	(Kiss et al., 2016, Kiss et al., 2021)	25
AMD	24,553 / 0	2010-2016	X	√	X	2017	(Wei et al., 2017, ArgusLab 2020)	217
CICAAGM2017	400 / 1500	2015-2016	√	X	dynamic	2018	(Lashkari et al., 2017, U. of New Brunswick 2020)	37
CICAndMal2017	426 / 5,065	2015-2017	√	√	hybrid	2018	(Lashkari et al., 2020, U. of New Brunswick 2020)	37
CICInvesAndMal2019						2019	(Taheri et al., 2019, U. of New Brunswick 2020)	19
CICAndMal2020	200k / 200k	N/A	√	X	static	2020	(Rahali et al., 2020, U. of New Brunswick 2020)	-
CICMalDroid2020	17,341 / 0	2017-2018	√	√	hybrid	2020	(Mahdavifar et al., 2020, U. of New Brunswick 2020)	-

small fraction of the existing Android malware families. *AMD* is composed of 24,553 malware samples belonging to 71 malware families and no benign samples. The *Kharon* dataset provides advanced static analysis data for 20 samples of representative malware families. It is too small for machine learning purposes, where data quantity matters. The rest of the datasets included in Table 1 are provided by the Canadian Institute of Cybersecurity (CIC). Some of the *CIC* datasets include samples of other datasets, such as *CICAndroidBot* which is composed of *MalGenome* dataset plus additional Android botnet samples. With the exception of *CICAndroidBot*, the rest of the datasets provided by the *CIC* only include recent malware at the time of the publication of the dataset and except *CICAndMal2020* and *CICMalDroid2020*, they are too small to provide complete coverage of the existing Android malware families. Although there is no time frame provided for *CICAndMal2020* dataset, its size makes it a good option for ML-based malware detection systems. However, as the *apk*s are not provided, the potential of this data set is limited to static feature-based machine learning detectors.

The majority of the datasets in Table 1 are publicly available or upon request and provide the original *apk* files, useful for further analysis and comparison. Nevertheless, at the time of writing this article, *AMD* and *MalGenome* are discontinued projects, making their data not accessible anymore in any format. This fact restricts the options for *old* malware samples to *Drebin* dataset.

As reported in Table 1, the most frequently collected features are *static*, especially in the *old* datasets, while *dynamic* features are just provided in some more recent datasets. Static features are easier and faster to collect what makes them more appealing and suitable for large data sets, such as the *CICAndMAl2020* dataset. The collection of dynamic features requires the usage of real devices or Android emulators, a decision that may also have an impact on the quality and quantity of the collected data (Guerra-Manzanares et al., 2019).

As a result, based on the information contained in Table 1, most of the datasets are considerably small, focused on static features, and span a short time-frame, usually 1-2 years. Only the *AMD* dataset studied the malware evolution as a concept in a time-frame spanning 6-7 years. The rest of the data sets do not take the *time* effect into consideration, restricted to small time periods, thus neglecting the importance of changes and evolution in Android malware over time. Ignoring this fact can harm significantly the performance of malware detectors over time, as malware evolves the importance of features to discriminate them effectively using ML techniques may change, a phenomenon called *concept drift*. Furthermore, most of the detection methods published in recent years use *old* malware data sets to induce and test their solutions, which might damage the generalization capabilities of the proposed models to recent malware.

Table 2 – Android malware repositories.

Repository	Composition	Date	Ref.
VirusTotal	+20,000 / 0	2014-2020	(VirusTotal 2020)
VirusShare	+40,000 / 0	2008-2020	(VirusShare 2020)
AndroZoo	+15 million?	2008-2020	(Allix et al., 2016, du Luxembourg, 2021)
Contagio Mobile	357 / 0	2011-2018	(Parkour, 2019)

The vast majority of the datasets provided in Table 1 provide an obsolete (i.e., old) and partial (i.e., a small amount of data and just static analysis) depiction of the Android malware phenomenon, insufficient to study the evolution of Android malware in general and malware families in particular. Although they remain as an important source of malware, especially regarding old samples, all the datasets in Table 1 must be complemented with other datasets or sources of malware in order to get the largest, widest, and most complete picture of the Android malware history and evolution. For such purpose, Table. 2 provides accessible general malware repositories that also contain Android malware samples. These repos-

itories are a remarkable source of malware that have already been used to complement and enrich existent datasets for research. Mainly designed as database services, they are growing repositories of malware samples. More specifically, *VirusTotal* and *VirusShare* are upon request malware repositories, *Contagio* is a discontinued but publicly available malware project and *AndroZoo* is a large repository of Android applications, but with unknown proportions of malware and benign apps.

Lastly, as can be extracted from the summary provided in Table 1, the majority of datasets just provide malicious apps, excluding benign samples. This fact restricts the possibilities of building effective machine learning-based classifiers using these datasets, as both types of applications are needed to build effective malware classifiers (i.e., supervised machine learning).

1.2. *Research Objective*

This research addresses the highlighted issues of existing datasets by merging different data sources to generate a large dataset with samples encompassing all years of Android history (i.e., 2008-2020), collecting static and dynamic features, and being the first Android dataset attending to the particularities of distinct dynamic data sources (i.e., real devices and emulators). Furthermore, it is also the first data set to include the *time* variable into the Android malware detection issue. As Android malware is a non-stationary phenomenon, it is an alive and constantly evolving phenomenon that should be placed into its temporal context. This research aims to fill this important gap, neglected by the available datasets, providing different sources of time information (i.e., timestamps). The time constraint might also be important as changes in Android OS (i.e, new OS releases) can affect the behavior and features used by the applications (e.g., the introduction of new permissions or their deprecation). Even though there is no straightforward method to ascertain the exact date of an Android application, it can be approximated using several techniques and tools, which are explored in this study.

1.3. *Contribution and Novelty*

As the main contribution of this research, we introduce *KronoDroid*, an Android malware data set that merges and complements data sources, covering an extensive period of time and characterizing each sample with static and dynamic features. *KronoDroid* aims to fulfill the gaps in the existing datasets and research, providing the following key aspects:

1 **Hybrid analysis data**. The combination of static and dynamic features provides more complete information about the applications.
2 **Distinct timestamp approaches for temporal context**. Different timestamp options are analyzed and proposed. As time is taken into consideration, it allows to learn about the evolution of malware and build more effective, robust, and long-lasting machine learning systems.
3 **Malware and benign labeled samples**. Both categories of applications are provided, which can be used to build effective classifiers and perform thorough application forensic analysis.
4 **Device-based behavioral data**. Existing dynamic data differences were found when applications are run in real or emulated devices. This dataset provides both sources of dynamic data. *KronoDroid* is composed of two equally-featured sub-datasets (i.e., real device and emulator) which can be used for comparison and further investigation.
5 **Large data set**. Over 28,000 samples per class on both sub-datasets.

The application of machine learning to Android malware detection is a widely-studied area. However, machine learning research studies still suffer various pitfalls including the inaccuracies related to datasets and their labeling (Arp et al., 2020), acting as a barrier to overcome real-world operational challenges such as adapting to evolving behavior, utilization of data source variations, and conducting family-oriented characterization. Our novel dataset has a big potential to facilitate the research in such directions. Our study can be considered as a large-scale data generation effort based on various key findings from our previous research (Guerra-Manzanares et al., 2019; Guerra-Manzanares et al., 2019; Guerra-Manzanares et al., 2019). We identified that the same dynamic or static features extracted from the apps belonging to different time-frames have varying discriminatory power, leading us to focus on the concept drift problem and collecting a relevant dataset (Guerra-Manzanares et al., 2019). We found out that the real devices and emulators may show variations in dynamic behavior that cause reduced detection performances in machine learning models if the type of data source is not considered (Guerra-Manzanares et al., 2019). Thus, we generated a dataset obtained from real devices in addition to emulators. We determined the optimum duration for collecting dynamic data according to the experimental results given in Guerra-Manzanares et al. (2019).

The paper structure is as follows: background information about Android malware analysis and literature review are provided in Section 2. The methodology implemented in this research is outlined in Section 4 while Section 5 shows a comprehensive analysis of the main outcome of this research, a novel Android malware data set. Section 6 provides the discussion points while Section 7 wraps up the study, highlights its major contributions, and establishes future work.

2. Background Information

This section provides the fundamental aspects of Android malware analysis and background information about the basic structure of Android applications.

2.1. *Structure of Android Apps*

2.1.1. *APK Bundle Structure*

On Android, everything the user interacts with, from the contacts list to games, is an app. Once compiled, apps are distributed as *Android package* (APK) files, a compressed ZIP archive identified by the *.apk* extension (Android 2021). The *APKs*' inner folders and file structures are consistent across applica- tions, enabling them to run in compatible Android devices (Android 2021). The *AndroidManifest.xml* and

the *classes.dex* files provide most of the relevant data about the app. The former declares all the essen- tial information about the app (i.e., more in Section 2.1.3) while the latter contains the compiled code (i.e., *Dalvik* bytecode) executed by *Dalvik/Android Runtime* virtual machine at runtime.

2.1.2. *Components: APK Building Blocks*

At a lower level, applications are designed and built on four main building blocks or *components*, which are used as entry points by users and the OS to interact with the app (Android 2021). Namely, *activities, services, broadcast receivers* and *content providers*. An *activity* is a single screen with a user interface where the interaction between the user and the app occurs (Android 2021). A *service* enables to keep running the app in the background (Android 2021). A *broadcast receiver* allows apps or the OS to deliver events to apps outside of the regular user flow (Android 2021) while a *content provider* manages a shared set of app data, granting other apps a secure way to access it (Android 2021).

Activities, services, and broadcast receivers are activated via asynchronous communication objects called *intents* (Android 2021), declared in the *AndroidManifest.xml* and handled by the OS. Content providers are activated using distinct mechanisms (Android 2021).

2.1.3. *The AndroidManifest.xml File*

The *AndroidManifest.xml* is a critical and the only mandatory file that every Android app must include. It contains all the essential information about the app so that the OS can manage it properly. Among other information, it declares (Android 2021; Android 2021):

- **Package name and version** - which uniquely identify the application.
- **App components** - to get the system acquainted with all the components of the app and enables them to be started.
- **User permissions** - requested security permissions needed by the app to perform tasks.
- **API level** - minimum API level required by the app to run on an Android system.
- **Hardware and software features** - used or required by the app to perform any of its tasks.
- **API libraries** - libraries needed by the application to run, distinct from the Android framework APIs.

The *AndroidManifest.xml* file contains critical information about the app that is used by the OS, *Google Play* and *Android build* tools to get and provide information about it (e.g., *compatibility* requirements). All this data can be collected using specific tools and be used for forensics analysis and malware detection.

2.2. *Android Malware Analysis*

There are two fundamental approaches for malware analysis: *static* and *dynamic*. Both approaches can be further categorized as *basic* or *advanced* and are briefly described as follows (Sikorski and Honig, 2012; Dunham et al., 2015).

- **Static** - involves the analysis of the malware file without being executed. The *basic* ap- proach involves the examination of the file without inspecting the actual code, to determine if the file is malicious and get its basic functionalities (e.g., antivirus check) while the *advanced* methods require *reverse-engineering* the malware internals through a deep examination of the malware code, thus providing exact information about the malware actions.
- **Dynamic** - the analysis is performed executing the malware in a controlled environment. In the *basic* approach, the malware is run in a *sandbox* and its behavior is observed, helping to produce effective signatures to prevent its spread, whereas the *advanced* methods imply the usage of other tools (e.g., *debuggers*) to examine the internal state of the running malicious file, providing detailed information about its behavior at runtime.

When both approaches are combined, **hybrid** analysis, they yield a more complete analysis of the sample and retrieve important complementary data. This is the approach used in this research where advanced static and dynamic techniques were used together to provide a better and more complete profile about each app behavior and functionalities.

3. Related work

This section provides a literature review about Android malware detection using machine learning techniques.

Android malware analysis approaches allow the collection of different types of features, used to characterize and build machine learning systems. Three main approaches are used, depending on the nature of the collected features: static, dynamic, or hybrid analysis (Feizollah et al., 2015). This section provides a concise review of the approaches and common features used for machine learning-based Android malware detection and introduces the *concept drift* related studies and data sets in the study field.

3.1. *Android Malware Detection*

3.1.1. *Static Android Malware Detection*

Static features from Android apps are mainly collected from two sources: inspection of the disassembled code and data extraction from the AndroidManifest.xml file. While some studies combine both feature sources (Arp et al., 2014; Felt et al., 2011; Li et al., 2018; Li et al., 2020; Peiravian and Zhu, 2013; Yerima et al., 2015; Wang et al., 2019), the majority sticks to a single source. In this regard, some solutions use code features from the disassembled code such as the program flow or the API function calls (Zhu et al., 2017; Grace et al., 2012; Yang et al., 2021; Cai et al., 2021; Hou et al., 2017) while from the AndroidManifest.xml the most used features are permissions (Peng et al., 2012; Enck et al., 2009; Talha et al., 2015; Liang and Du, 2014; Mcdonald et al., 2021) and intent filters (Feizollah et al., 2017), alone or combined to boost detection performance (Idrees and Rajarajan, 2014). In brief, API calls indicate the functions called by the app in its source code, security permissions the degree of privilege the app requests to access some type of data (i.e., sensitive data) and intent filters

denote the actions the app is intended to perform. Static analysis features are easy to collect and provide extensive code coverage but can easily avoid detection when code obfuscation techniques are used. Data encryption, update attacks, obfuscation, or polymorphic techniques are used to hide the malicious code to bypass static features-based malware detection systems (Alzaylaee et al., 2020).

3.1.2. *Dynamic Android Malware Detection*

Dynamic features are collected when the application is interacting with the operating system or the network. System calls (Burguera et al., 2011; Guerra-Manzanares et al., 2019; Hou et al., 2016; Tam et al., 2015; Guerra-Manzanares et al., 2019) and network flow (Lashkari et al., 2017; Arora et al., 2014) are the most common features used (Feizollah et al., 2015). Less commonly, CPU and RAM usage, running processes, battery statistics, API function calls and other runtime features have also been used alone (Schmidt, 2011; Enck et al., 2014; Amos et al., 2013) or combined with system calls or network packets (Dini et al., 2012; Shabtai et al., 2012). Briefly, system calls are used for app-OS communication while network flow is obtained from the app-network interaction. Even though they can be bypassed (Petsas et al., 2014), dynamic features-based detection methods are robust to code obfuscation and encryption techniques. However, they are more time-consuming and difficult to collect as they require the app to be installed and run in a sandboxed device, either an emulator or a real device. Both kinds of devices have been used in the literature, assuming that, disregarding the device influence, the app behavior would not reflect any change in the dynamic features collected. When the dynamic approach is used, greater code coverage can be achieved when using pseudo-random user-generated events (Alzaylaee et al., 2020).

3.1.3. *Hybrid Android Malware Detection*

The combination of static and dynamic features collected from applications is used by a smaller proportion of the existing research (Grace et al., 2012; Alzaylaee et al., 2020; Kabakus and Dogru, 2018; Guerra-Manzanares et al., 2019; Yuan et al., 2014; Bl¨asing et al., 2010). However, these approaches tend to provide more complete information about the malware as they collect the application's runtime behavior complemented by relevant static features.

As can be observed, the predominant trend in the studies is to use either static or dynamic approaches, thus neglecting the potential of their combination (i.e., hybrid approach). As a contribution to the field, this research focuses on the collection of hybrid features of the generated dataset for Android malware detection. Hybrid analysis constitutes a smaller part of the existing research (Feizollah et al., 2015). It is more complex and time-consuming to perform but enables to generate a better overall picture of the problem at hand, where the usage of complementary data may yield better results (Guerra-Manzanares et al., 2019).

3.2. *Android Malware Concept Drift*

3.2.1. *Related Studies*

The vast majority of the previous studies do not take the *time* variable into account, considering malware as stationary data. Thus neglecting the changes in malware over time and their degenerative impact on the performance of the machine learning-based detection methods, a phenomenon called *concept drift*. This is also emphasized by the fact that the vast majority of the studies published in recent years use *old* malware data sets to test and prove their findings. Their results may not generalize to recent malware as malware is a constantly evolving phenomenon that must be placed into a temporal context to be fully comprehended.

This approach is considered in the studies summarized in Table 3, which provides a detailed overview of the recent works that addressed the phenomenon of *concept drift* in Android applications. The *Time-frame* attribute for each study (i.e., *Name/Ref*) provides the temporal window of the applications analyzed, while the size of the dataset and the malware source are provided in the column *Dataset Size* (i.e., B for benign data size, and M for malware data size) and *Dataset Source* respectively. The *Features* column provides information about what kind of data was used to characterize the samples on the study period. The *Timestamp* states how the applications temporal context (i.e., date) was determined while the *Performance* shows the reported performance metric of the solution. Finally, the *Year* column establishes the publication year of the research.

As can be noticed, the time-frame encompassed in the studies included in Table 3 varies significantly. While there are some studies that focus on relatively narrow time-frames (i.e., 8 months in the case of *DroidOL*), the majority encompass a significant amount of years (i.e., an average of 6.6 years) being *AndroCT* the most extended, ranging from 2010 to 2019. In this research, the collected dataset doubles the average time of these studies, providing samples from 2008 to 2020 (i.e., 13 years data), which allows to analyze the issue in longer time-frames and contrast the obtained results.

Regarding the datasets, the most prevalent dataset is AndroZoo, followed by Drebin, an *old dataset*. The legitimate samples come mainly from Google Play. In this research, we used Drebin as a source of *old* malware data and other more recent well-known malware sets to constitute the body of malware data. The usage of well-known and established data sets (i.e., Drebin, AMD, VirusTotal, and VirusShare) was preferred to the usage of AndroZoo, which is a good source of applications in general (i.e., over 15 million apps) but not specifically of malware instances, thus needing to rely exclusively on the antivirus (AV) detection report to label the apps from the repository. We used the AV detection report as extra support for the label and we required the original data source to provide the label, as explained in Section 5.2. The size of the datasets varies significantly among the studies, being the one used in DroidOL the largest one in terms of malware apps. Our dataset is composed of a similar amount of malware instances (i.e., in the real device case), providing a large corpus of data for malware and similarly of legitimate apps.

As can be observed, all the studies focus on the analysis of API calls in a dynamic or static analytical methodology. API function calls are eminently static objects which can also be acquired dynamically or *traced* when the app is executed, providing a behavioral perspective complementary to the static approach. This is the approach taken by the studies marked by an asterisk (*) in Table 3. Function calls provide a dynamic

Table 3 – Recent Android malware research works that take into account the time.

Name/Ref	Time-frame	Dataset		Features (#)	Timestamp	Performance	Year
		Source	Size				
		Google Play Anzhi AppChina SlideMe HiApk FDroid	B: 44,347	Graph-kernel			
DroidOL (Narayanan et al., 2016)	2014	Angeeks	M: 42,910	(1,653,496)	Creation day	Acc: 0.84	2016
ENBCS (Hu et al., 2017)	N/S	Drebin Other N/S	B: 5,101 M: 1,761	Permissions (152) API calls (24) Actions (229)	N/S	Acc: 0.96	2017
TRANSCEND (Jordaney et al., 2017)	2010-2014	Drebin MARVIN	B: 133,127 M: 14,739	Permissions API calls Other static	Train: Drebin Test: MARVIN	Prec: 0.89 Rec: 0.76	2017
(Cai and Ryder, 2017)	N/S	Google Play	B: 125	API calls*	N/S	N/A	2017
(Cai and Ryder, 2017)			M: 0	(selected, 122)			
MamaDroid (Onwuzurike et al., 2019)		Drebin	B: 8,447				2016
MamaDroid ext. (Mariconti et al., 2016)	2010-2016	Virushare	M: 35,493	API calls	N/S	0.87	2019
DroidCat (Cai et al., 2019)	2009-2017	Google Play AndroZoo VirusShare Drebin MalGenome	B: 17,365 M: 16,978	API calls* (selected, 122)	First seen VT (range of years)	F1: 0.97	2018
(Cai and Jenkins, 2018)	2012-2017	N/S	B: 3,431 M: 3,001	API calls* (selected, 122)	N/S	F1: 0.82-0.93	2018
DroidEvolver (Xu et al., 2019)	2011-2016	AndroZoo	B: 33,294 M: 34,722	API calls	Compilation time (dex date, year)	F1:0.95-0.85	2019
(Fu and Cai, 2019)	2010-2017	N/S	B: 13,627 M: 11,153	API calls* Other N/S	N/S	F1: 0.71	2019
EveDroid (Lei et al., 2019)	2012-2018	PlayDrone Google Play VirusShare	B: 14,956 M: 28,848	API calls	VT Submission (Year)	0.99-0.84	2019
TESSERACT (Pendlebury et al., 2019)	2014-2016	AndroZoo	B: 116,993 M: 12,735	Permissions API calls Other static	Compilation time (dex date, year)	F1: 0.91-0.82	2019
TRANSCENDENT (Barbero et al., 2020)	2014-2018	AndroZoo	B: 232,848 M: 26,387	Permissions API calls Other static	Compilation time (dex date, year)	F1: 0.90-0.70	2020
APIGraph (Zhang et al., 2020)	2012-2018	Google Play AndroZoo VirusShare VirusTotal AMD	B: 290,505 M: 32,089	API calls	Appearance (Year)	F1: 0.92-0.68	2020
(Cai et al., 2020)	2010-2017	Google Play VirusShare AndroZoo	B: 15,451 M: 15,183	API calls*	Compilation time (dex date, year))	N/A	2020
DroidSpan (Cai, 2020)	2010-2017	Google Play VirusShare AndroZoo	B: 13,627 M: 12,755	API calls*	N/S	F1: 0.92-0.72	2020
(Cai, 2020)	2010-2017	N/S	B: 1,000 M: 1,000	API calls*	N/S	N/A	2020
(Cai and Ryder, 2020)	2010-2017	Google Play AndroZoo	B: 3,000 M: NA	API calls* (122)	N/S	N/A	2020
AndroCT (Wen Li and Cai, 2021)	2010-2019	Google Play VirusShare AndroZoo	B: 18,277 M: 17,697	API calls* (122)	N/S	N/A	2021

N/A - Not applicable, the evaluation was not performed or provided N/S - Not specified
* The feature was collected dynamically

behavioral profile of the app which is prone to changes due to API/libraries modifications, deletions, or additions. The approach taken in this study focuses on the usage of a distinct dynamic object, *kernel* or *system calls*, which are eminently dynamic objects (i.e., they cannot be traced in a static way as API calls) that provide a robust behavioral profile of the application, less subtle to drastic changes along years. This provides a distinct and novel approach with regard to all the studies in Table 3. While *API calls* and *system calls* can both provide an app's behavioral profile, they are not directly linked as they occur in different OS regions (i.e., *user* vs. *kernel* space) and they are issued with distinct purposes (i.e., kernel calls are requests to the OS while API calls are API framework *function calls* to perform specific tasks that may or may not trigger a *kernel call*). As a result, API function calls and system calls provide a distinct approach to the same study object. In addition to that, a novelty provided by the *KronoDroid* dataset is the usage of additional static features such as permissions, intents, etc. to enhance malware detection and Android app comprehension from a hybrid perspective. None of the studies in Table 3 combine both approaches using distinct objects for app characterization.

The temporal context approach and the usability of the dataset are distinctive points of *KronoDroid* when compared to the studies given in Table 1 and Table 3. Regarding the temporal context, the studies that consider the temporal dimension attach themselves to a particular timestamp, generating a single perspective that may be prone to temporal bias and errors (e.g., *dex date* timestamp used in many studies is no longer reliable as the majority of apps have it set at 1980 (du Luxembourg, 2021)). In order to overcome this limitation, *KronoDroid* provides 4 possible timestamps per sample, depending on the temporal context source, helping to provide a more exact temporal context for the studied apps. Regarding usability, *KronoDroid* is provided in a ready-to-use tabular format (i.e., CSV) which can be directly used without having to process *log* files or the *raw* data. However, this latter option is also provided for more technically experienced individuals. Therefore *KronoDroid* can be used by any interested researcher without the need of deep technical data extraction knowledge.

In this study, for the sake of completeness, taking into account the different dynamic malware analysis devices used in research, the dynamic features of this data set were collected using two devices: an emulator and a real device. In this regard and even though researchers tend to assume that the behavior of an app does not change according to the platform or device used, our experimentation and research indicate that specific dynamic features, such as system calls, may change significantly according to the device type where they are collected (Guerra-Manzanares et al., 2019). As a result, this research aims to contribute to a deeper exploration of these differences by providing kernel-related dynamic features (i.e., system calls) collected on two different Android platforms (i.e., ARM and x86 devices).

3.2.2. *Related Datasets*

The datasets used in concept drift studies in Android malware detection are provided in Table 4. This table reflects the datasets made publicly available or on request by the studies contained in Table. 3. This concise table specifies the name of the dataset (or the related study), its composition (i.e., *Size* column; *B* for benign data and *M* for malware data; *E* and *R* are added to specify the dataset if Emulator and Real device datasets have different compositions), the time-frame they encompass (i.e., *Time range*), the availability of the dataset and its format, the malware analysis methodology used to collect the dataset features (i.e., static or dynamic), the data source when the dynamic approach was used (i.e., emulator or real device), the type and number of features provided by the dataset and the *reference* to the dataset repository.

As can be noticed, although some studies included in Table 3 released the hashes of the data samples they used thus enabling reproducibility of the results (Zhang et al., 2020), just a minority of the studies released their data sets for public use.

When comparing *KronoDroid* with the available datasets for concept drift research, provided in Table 4, it can be argued that *AndroCT* (Wen Li and Cai, 2021) is similar, as they are the only ones providing data tested on emulators and real devices. However, even though the features of AndroCT make it an interesting and useful Android dataset, KronoDroid is significantly distinct from it in both the perspective used and its broad comparative metrics. Regarding the perspective, KronoDroid uses dynamic (i.e., system calls) and relevant static features such as permissions (i.e., 489 in total, including timestamps) to characterize each application while AndroCT provides just dynamic data (i.e., API calls). This fact makes KronoDroid usable for a wide variety of researchers and malware detection approaches, choosing freely the focus on one or another perspective. When the broad comparative metrics are analyzed, the size and time-frame encompassed by KronoDroid are unprecedented. Composed by more than 28,000 per class (i.e., benign/malware), including over 200 malware families, and encompassing 13 years (i.e., 2008-2020) of Android history make it, to the best of our knowledge, the largest, fully labeled and most extended timestamped Android dataset. Lastly, whereas AndroCT provides all *log* traces and collected data in a semi-processed format, KronoDroid is readily available in structured tabular format (i.e., CSV file) and *raw* format (i.e., not processed *log* files), matching the distinct types of technical needs and capabilities that machine learning practitioners have.

In conclusion, the present dataset aims to address the aforementioned research gaps and improve the field of Android malware detection. However, this research is not free of limitations, weaknesses, and challenges, which are covered in Section 6.1 after the methodology and results are explained in the following sections.

4. Methodology

KronoDroid, the major contribution of this research, is a hybrid-featured Android malware dataset that introduces the time feature in the Android malware analysis. The acquisition of the hybrid features and the timestamps required the utilization of static and dynamic analysis techniques on the collected samples. The features collected are detailed in Table 7 whereas the hybrid analysis workflow is depicted in Fig. 1 and is thoroughly explained in the following paragraphs.

Table 4 – Android malware repositories.

Name	Size	Time Range	Availability	Analysis	Source	Features (#)	Reference
MaMaDroid	B: 8,402 M: 34,521	2010-2016	On request	Static	-	API Calls (-)	(Stringhini, 2018)
TraceDroid	B: 15,451 M: 15,183	2010-2017	Public (Raw data)	Dynamic	Emulator	API Calls (122)	(Cai, 2020)
AndroCT	B: 18,277 M: 17,697	2010-2019	Public (Raw data)	Dynamic	Emulator Real device	API Calls (122)	(Li et al., 2021)
KronoDroid	BE: 35,246 BR: 36,755 ME: 28,745 MR: 41,382	2008-2020	Public (Raw data and CSV)	Static Dynamic	Emulator Real device	Syscalls (289) Permissions (166) Other static (34)	(Guerra-Manzanares, 2021)

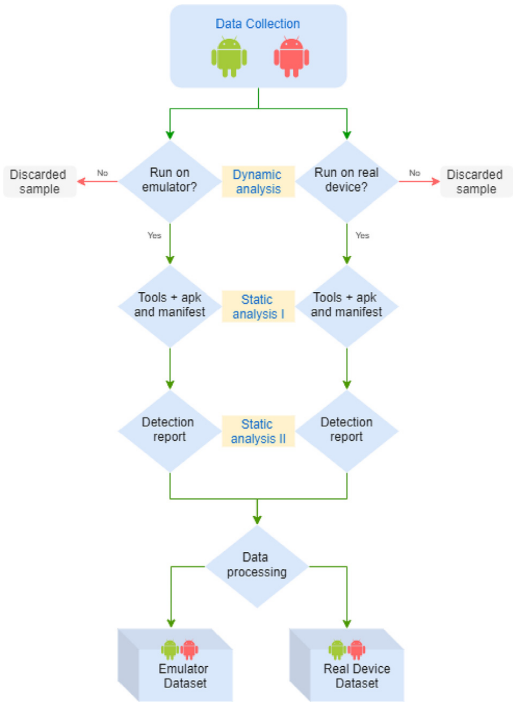


Fig. 1 – Data set generation workflow.

Table 5 – Initial malware dataset composition.

Data set	Size	Time-frame
Drebin	5,560	2010-2012
AMD	24,553	2010-2016
VirusTotal	21,687	2014-2020
VirusShare	3,034	2008-2018
Total	54,834	2008-2020

Table 6 – Initial benign dataset composition.

Data set	Size	Time-frame	Ref.
F-droid	6,919	2010-2020	(F-droid 2020)
MARVIN	29,941	2008-2014	(Lindorfer et al., 2015)
APKMirror	2,200	2010-2020	(APKMirror 2020)
Total	39,060	2008-2020	-

4.1. *Data Collection: The Workflow*

The first step of the data collection workflow involved the acquisition of the Android samples which composed the *initial* dataset. The *initial* dataset is composed of 93,894 Android applications that were acquired from 7 different datasets and repositories. After the collection of the initial dataset, dynamic analysis was performed using two different Android devices: an emulator and a real device. The rationale for the usage of these two devices is extracted from Guerra-Manzanares et al., 2019. According to the authors, the registered dynamic behavior of Android applications might differ across platforms when using system calls as features. This inconsistency may cause a great unexpected impact in the induced machine learning models (i.e., models not generalizing to both sources of data). Thus, for the sake of completeness, as both types of devices are used in the Android malware detection research literature without any distinction, this research was performed using both emulator and real device platforms. Consequently, after the dynamic phase, two distinct platform-based data sets were generated from the initially collected set of applications. As malware and benign Android applications were collected, two app *categories* were generated (i.e., also referenced as *classes* or *labels*). The composition of each *initial* single-class dataset is provided in Table 5 for malware apps and Table 6 for benign applications. As can be observed, the overlap of the time-frames of the applications included in both datasets encompass from 2008, when the first commercial Android version was released, until 2020. After the dynamic analysis phase, two sequential static analysis phases, referred as *static analysis I* and *static analysis II* in Fig. 1, were performed before all the collected data were processed and the features generated, thus conforming the two *final* platform-based datasets. The whole process is detailed in the following paragraphs.

4.1.1. The Malware Data

The *initial* malware dataset is composed of applications belonging to two malware datasets and two repositories, outlined in Table 5. More specifically, *Drebin* is a malware dataset collected between 2010-2012 and presented in Arp et al., 2014, including 5560 samples from 179 malware families. It is the most used academic dataset for Android malware research nowadays (Irolla and Dey, 2018). Cited more than a thousand times since its generation, it has become the replacement of the discontinued MalGenome Project dataset (Zhou and Jiang, 2012). *Drebin* provides the *older* malware samples for this study (i.e., belonging to the early years of Android history). The *Android Malware Dataset* (Wei et al., 2017), is a public dataset composed of 24,553 malware samples, split into 135 varieties among 71 malware families and collected between 2010 and 2016. The *VirusTotal Academic Malware Samples* dataset is a malware repository collected by *VirusTotal* using their antivirus engine (VirusTotal 2020). It is a growing repository generated by *VirusTotal* from the positively detected files submitted for scanning by its users. Its collection time-frame ranges from 2014 to 2020. At the time of this study, the repository had 21,687 Android apps. Finally, *VirusShare* (VirusShare 2020) is a repository of malware samples that provides access to *live* malicious samples for research and forensics purposes. For this study, 3,034 Android applications were randomly selected and downloaded from *VirusShare*.

As reported in Table 5, *old* malware data are covered using *Drebin* and *AMD* datasets while more recent malware is covered by apps belonging to *VirusTotal Academic Malware Samples* dataset. The set of applications from *VirusShare* are added for the sake of completeness of the dataset, as they encompass the widest time-frame. Consequently, the malware data covers the widest time-frame possible, from Android OS first release, 2008, to the present time, 2020.

4.1.2. The Benign Data

The malware dataset is composed of 54,834 applications while the *benign* data set includes 39,060 samples. This difference is due to the more challenging task of collecting *old* benign applications.

Unlike malware apps, legitimate apps are not usually stored or available in repositories, neither provided in the existing malware datasets, as reported in Table. 1. Also, as they are frequently updated and adapted to OS releases by their developers, they tend to generate more *compatibility* issues than malware apps (i.e., requiring a higher minimum API level). Furthermore, as the legitimate alternative *app markets* started their activity some time after the release of Android, their app set is significantly limited for the first years of Android history. Therefore, due to the focus on malware and scarcity of benign data, the *initial* benign dataset was out-numbered by the *initial* malware dataset.

The *initial* benign dataset was generated from three sources, provided in Table. 6, two public Android app *markets* and a research dataset. In this regard, *F-droid* (F-droid 2020) is a free, open-source Android repository that provides apps to users via a client software. It is a well-known alternative to the official Android apps market (i.e., *Google Play*) that performs security checks to ensure the apps provided are malware-free. At the time of this research, the F-droid repository contained 6,919 samples, ranging from 2010 to 2020. Similarly to F-droid, *APKMirror* is a larger, web-based Android alternative app store (Apkmirror 2020). Due its security checks, it is considered a trusted and secure source of apps (APKMirror 2021). For this research, 2,200 randomly selected apps, uploaded to the app catalog between 2010 to 2020, were used. Finally, 29,941 samples from *MARVIN* benign dataset (Lindorfer et al., 2015), shared by the authors, were used in this study. They allow covering the early years of Android history, from 2008 to 2014.

Similar to the malware dataset, the overlap of the data sources enables complete coverage of the whole 2008-2020 period, as shown in Table 6. More specifically, *old* benign apps were provided by *MARVIN* dataset while recent benign samples were provided by *F-droid* repository. *APKMirror* samples were used for the sake of completeness and time-frame coverage, especially for the recent years.

4.2. Phase II: Dynamic Analysis

System or *kernel* calls are the most used behavioral feature in dynamic malware detection research and the dynamic features collected in this study. System calls are the fundamental interface between the apps and the operating system kernel. They provide a level of abstraction and security, acting as handlers of service and resource requests from the applications using API (i.e., user-level) to the OS (i.e., kernel-level) (Bovet and Cesati, 2005). Android OS is built on top of the Linux kernel, which provides more than 200 system calls, depending on the CPU architecture (e.g., ARM or x86). More specifically, kernel calls manage the requests of services and resources that the apps are not allowed to perform directly, related to process control, file management, device management, communication, etc. Therefore, system calls analysis provide *dynamic* data about the real behavior of apps at runtime.

The system calls collection in this study was performed by means of an automated script using *Android Debug Bridge* (ADB) commands. *ADB* is a command-line tool that enables the communication between a computer and a mobile device. The script, developed by the author, attempted to install, execute, monitor, log, and uninstall all the applications composing the *initial* datasets in each of the devices used. Two devices were used for this research: a Samsung A6 (i.e., real device) and a Nexus 5X Android SDK emulator instance (i.e., emulated device). The same Android version (i.e., Android 8.0 *Oreo*), configuration, *script*, *initial* dataset and acquisition procedure was in place for both devices.

As Android applications can depend on specific libraries to be executed in different CPU architectures and require a minimum API level (i.e., compatibility issues), not all the apps from the initial dataset were successfully installed on both devices (Android 2020). For instance, if the application had only native libraries for ARM-based devices (i.e., armeabi), it was not successfully installed in the emulator (i.e., x86 architecture), deemed as *incompatible* app. The set of incompatible apps for each device were discarded for any posterior steps, not further processed, and did not form part of the *final* dataset for the specific device they were not compatible with.

Regarding the compatible apps, after their successful installation, they were attempted to boot up by invoking the application main activity using the *monkey* tool (Android 2021).

If this step was successful, they were allowed to run freely, without any further interaction, for 60 seconds. The rationale behind this run-time constraint is based on the findings of Guerra-Manzanares et al., 2019, where different run-time time-frames were evaluated in a similar dataset and the authors concluded that longer time-frames (i.e, up to 15 minutes) did not lead to improved detection performance. During the run-time, the application was monitored and the system calls issued by the main process were logged using *strace* tool (Levin, 2021). The generated *log* file, containing the issued system calls by the app at run-time, was pulled into the storage device using ADB before the application was uninstalled from the device.

As a result, only the successfully installed and executed applications are included in the *final* device- related labeled datasets. Therefore, the dynamic analysis acted as a filter stage, determining the apps that finally form part of either of the resulting datasets, thus ensuring the collection of hybrid analysis features for all the samples included.

4.3. *Phase III: Static Analysis*

After the dynamic analysis step, two sequential static analysis steps were performed on the remaining set of applications. These steps are outlined in the following paragraphs.

4.3.1. *Static Analysis I: APK Data Extraction*

The *apk* archive and the *AndroidManifest.xml* inside the *apk* container were analyzed and relevant information extracted.

From the *apk* bundle, metadata such as *filesize, timestamps* (i.e., *last* modification, *earliest* modification) and *SHA-256* hash were retrieved.

All Android applications must include the *AndroidManifest.xml* file inside their *apk* containers. This file contains relevant data and metadata, such as the app's package name, the components of the app (i.e., activities, services, broadcast receivers, and content providers), the required hardware and software features, and the security permissions the app needs to access protected resources (i.e., requested permissions) (Android 2021). In this first static analysis step, relevant features used by static malware detection studies were extracted from the *AndroidManifest.xml* of each processed application:

- **Android permissions**: all the requested permissions were extracted. The current list of Android *standard* permissions includes 166 permissions, categorized into 3 different levels of risk, also known as *protection levels*: *dangerous*, *signature*, or *normal* (Android 2021). As Android allows to define *custom* permissions, if any was declared, it was also collected.
- **Android Intent Filters**: all the intent filters data declared by the app were extracted.
- **Hardware & Software features**: hardware features and software requirements defined by the application were retrieved.
- **Other relevant data**: such as package name, activities, and services declared.

In this phase, all the collected data was retrieved using a *script*, developed by the author, by means of *AndroGuard, Android Asset Packaging Tool (aapt)* and *ExifTool*. More specifically, *AndroGuard* (Desnos et al., 2018) and *aapt* (Android 2021) were used to explore the *apk* archive and retrieve the relevant data from the *apk* and *AndroidManifest.xml* file. *ExifTool* (Harvey, 2021) was used to retrieve metadata about the *apk* bundle.

4.3.2. *Static Analysis II: Detection Report*

The second step of static analysis was performed using *VirusTotal* AV engine. *VirusTotal* is an antivirus scanner service which in its basic version allows the user to upload files and obtain a malware detection report and other related signals based on over 50 antiviruses' results (VirusTotal 2020). *VirusTotal API* was used to obtain the detection report and relevant metadata for all the apps filtered on the dynamic phase. As VirusTotal terms of agreements did not allow to share their collected timestamp data (i.e., first seen and first seen in the wild) nor other raw data from the report in the public dataset, they were not included as features in the released dataset. However, anyone interested in those features, used and discussed in this research, can collect them directly from the source, VirusTotal, using the *SHA-256* list and the *script* provided in the dataset repository.[1] More information is provided in Table. 7. In summary, this second static analysis step meets two purposes:

1. **Malware detection check** - in the case of malware samples, it provides data about the malware family and detection ratio. In the case of benign samples, the detection report is used as an additional check of their not malicious content. When allegedly benign apps were detected as malware by some AV, their real label was questioned and reflected in the dataset. The same is applied in the alleged malware found malware-free. For that purpose, the concepts of *soft* and *hard* labels are defined and implemented in this study. A detailed overview of them and the analysis results are reported in Section 5.2.
2. **Collection of relevant static data** - the detection report provides additional static features that are used to enrich the quality and completeness of the dataset, such as the detection-related timestamps.

4.3.3. *Phase IV: Data Processing*

After all the previous steps were performed, all the gathered and logged data for each application were further processed in order to extract and construct meaningful features to characterize each application. A total amount of 489 features were extracted and constructed to describe apps on both datasets (i.e., emulator and real device data) and each app class within them (i.e. malware or legitimate). Table 7 provides a concise description of the features that characterize each app. More specifically, apps are characterized by the following static features:

- **Package Name** - application ID or identifier string that uniquely identifies the appli- cation. It is defined in

[1] Script link: https://github.com/aleguma/kronodroid

Table 7 – Data set features summary.

Category	Count	Var type	Brief description
Package Name	1	Categorical	Application's package name
System Calls	288	Numeric	Absolute frequency of each syscall from the syscall set issued by the app at run-time. For consistency, the syscalls set is composed of 288 features
Total system Calls	1	Numeric	Total number of syscalls issued by the app at run-time
Standard Permissions	166	Categorical	Binary feature that indicates whether the standard Android permissions was requested (i.e., 1) or not requested (i.e., 0) by the app
Standard Permissions Categories	3	Numeric	Total number of normal, dangerous and signature permissions requested
Total Standard Permissions	1	Numeric	Total number of standard permissions requested by the app
Custom Permissions	1	Categorical	Binary feature that indicates if any custom permission was declared by the app (i.e., 1 if *any*, 0 if none)
Total Custom Permissions	1	Numeric	Total number of custom permissions defined by the application
Total Permissions	1	Numeric	Total number of requested permissions. Sum of standard and custom permissions
SHA-256	1	Categorical	SHA-256 hash value of the *apk* file
Compressed Filesize	1	Numeric	Compressed (apk) filesize of the app
Uncompressed Filesize	1	Numeric	Not compressed size of the file (i.e., sum of the size of all the raw *apk* inner files)
Timestamps	4	Numeric	Collected timestamps (i.e., *inner files* earliest and last modification, timestamp of first seen by VirusTotal* and first seen itw*)
Files	1	Numeric	Number of files inside the *apk* container (i.e., *inner* files)
Hardware & Software Features	1	List	List of hardware and software requested features
Activities	1	List	List of activities defined by the application
Total Activities	1	Numeric	Total number of activities defined by the application
Intent filters (IF) List	1	List	List of all the Intent filters defined by the application
IF Activities	1	Numeric	Total number of activities defined in intent filters
IF Activities Actions	1	Numeric	Total number of actions related to activities defined in intent filters
IF Services	1	Numeric	Total number of services defined in intent filters
IF Services Actions	1	Numeric	Total number of actions related to services defined in intent filters
IF Receivers	1	Numeric	Total number of receivers defined in intent filters
IF Receivers Actions	1	Numeric	Total number of actions related to receivers defined in intent filters
Total Intent filters	1	Numeric	Total number of intent filters declared by the application
Services	1	List	List of services defined by the application
Total Services	1	Numeric	Total number of services defined by the application
Detection Ratio	1	Numeric	Ratio of the AV scanners that detected the sample as malware. Value in the range [0,1].
Scan Time	1	Numeric	Timestamp that indicates when the app was scanned for this research
Malware Family	1	Categorical	Most likely malware family of the sample
Soft Label	1	Categorical	Soft label for the specific app. Possible values: {0,1}
Hard Label	1	Categorical	Hard label defined for the specific app. Possible values: {-1, 0, 1}
Total Features	489	-	313 Numeric, 172 Categorical and 4 List Features

* Features not provided with the dataset. More info at https://github.com/aleguma/kronodroid

the form of a Java package name or a reversed DNS domain name, aiming to avoid naming collisions (e.g., com.company.appname) (Android 2021). In this dataset, this single feature corresponds to the identifier string, which can be used to identify a particular app or different samples with the same package name in the whole dataset.

- **Standard Permissions** - permissions are a security mechanism that allows Android to limit what an app can access (e.g., contacts list) and do (e.g., record audio). They can be requested by the application either at installation time (normal permissions) or at run-time (dangerous permissions). Permissions requested by the application are declared in the *AndroidManifest.xml* file using the syntax: *android.permission.PERM*, where *PERM* is substituted by the corresponding permission name. For example, android.permission.INTERNET would be requested by the application to have connectivity, enabling it to open network sockets. A total quantity of 166 *standard* permissions are defined by the permissions API provided for Android developers (Android 2021). As a result, 166 categorical binary permission features were generated for each sample. The value for each feature can be either 0 or 1 (i.e., *unset* or *set* permission). A value of 0 reflects that the permission was not requested by the app while a value of 1 indicates a requested permission.

- **Permission Statistics** - Android categorizes permissions according to the *sensitive* information they can access and the security threat they may pose. Based on that, *normal, dangerous* and *signature* permissions are defined. *Custom* permissions can also be created and used. Consequently, 7 statistical features were computed for each application in relation to the permissions requested.
  - **Total Standard Permissions** - total quantity of *standard* permissions re- quested by the application.
  - **Total Dangerous** - total quantity of requested *standard* permissions that are categorized as *dangerous*. Indicative of permissions that request access to the user's personal data, which may pose a security threat.
  - **Total Normal** - total quantity of requested *standard* permissions that are categorized as *normal*.
  - **Total Signature** - total quantity of requested *standard* permissions that are categorized as *signature*.
  - **Custom Permission** - binary categorical feature that indicates whether any custom permission was declared by the application (i.e., value of 1). If no cus- tom permissions were defined the value of 0 is reported.
  - **Total Custom** - total number of *custom* permissions requested by the application.
  - **Total Permissions** - total number of permissions requested by the application (i.e., summation of all *custom* and *standard* permissions requested).
- **Compressed Filesize** - integer that reflects the *apk* file size in bytes.
- **Uncompressed Filesize** - integer that reflects the sum of the raw size of the applica- tion's *inner* files in bytes.
- **Timestamps** - temporal metadata about the application. As this information can be tam- pered or *wrong*, 4 distinct and complementary timestamps were collected, helping to generate the temporal context of the application. The 4 timestamp features collected are:
  - **Earliest Modification** - earliest modification timestamp found in any of the *in- ner* files of the application.
  - **Last Modification** - latest modification timestamp found in any of the files that compose the application.
  - **First Seen VT** - timestamp that denotes when the app was submitted for the first time to VirusTotal (VT) scan service.
  - **First Seen in the Wild (ITW)** - timestamp that indicates when the app was seen anywhere, online, for the first time. This feature was extracted from the VT scan report.

More specifically, *Exiftool* was used extract the *Earliest modification* and *Last Modification timestamps* from the files inside the *apk* while the VirusTotal report provided the *First Seen VT* and *First Seen ITW* timestamps.

- **SHA-256** - hexadecimal representation of the 256-bit hash string of each sample, uniquely identifying them. In malware analysis, SHA-256, a secure hash algorithm, is used to uniquely identify samples. It also helps to differentiate between distinct instances of the same malware, which may have slight modifications but the same package name.
- **Files** - numeric feature that reports the number of files inside the application's compressed container (i.e., *inner* files).
- **Activities** - expanded in two features. One provides the list of activities declared by the application (i.e., names) and the other provides the total number of them (i.e., count).
- **Intent Filters (IF)** - list of all the intent filters defined by the application.
- **Intent Filters Statistics** - 7 summary statistics are computed for each application in relation to the intent filters declared.
  - **Intent Activities** - total number of intent filters related to activities.
  - **Intent Activities Actions** - number of defined *actions* in the intent filters related to activities.
  - **Intent Services** - total number of intent filters related to services.
  - **Intent Services Actions** - number of defined *actions* in the intent filters related to services.
  - **Intent Receivers** - total number of intent filters related to receivers.
  - **Intent Receivers Actions** - number of defined *actions* in the intent filters related to receivers.
  - **Total Intent Filters** - total number of intent filters declared by the applica- tion.
- **Services** - expanded in two features. One provides the list of services declared by the application (i.e., names) and the other provides the total number of them (i.e., count).
- **Detection Ratio** - numeric value compressed in the range [0,1] that reflects the amount of AV scanners that positively detected the sample (i.e., as malware) over the total amount of scanners. It is a constructed feature based on information provided by the detection report (i.e., nr. of positive detections / nr. of scanners).
- **Malware Family** - most likely malware family of the sample according to the results provided by the AV scanners that positively detected the sample (i.e., majority of the vote). A detailed explanation of the procedure followed is provided in Section 5.4.
- **Scan time** - timestamp that reflects the time when the app was scanned by the VirusTotal AV engine, thus generating the parsed detection report.
- **Soft Label** - binary feature that indicates if the application is malware or be- nign according to the data source. A value of 0 denotes a benign application while a value of 1 indicates malware.
- **Hard Label** - binary feature that indicates if the application is malware or legitimate according to the detection report and its source. A value of 0 denotes a benign application, a value of 1 indicates malware and a value of -1 denotes a dubious sample. This label is explained in detail in Section 5.2.

From the dynamic perspective, the following features are defined for each application:

- **System Calls** - absolute frequency of each specific system call issued by the app during the run-time. As two distinct devices were used to collect these features, powered by different CPU architectures (i.e., x86 for the emulator, ARM for the real device), two distinct syscalls set were retrieved.

Even though x86 and ARM architectures have different system calls set, for the sake of consistency of the datasets, the union of both system calls sets was used to generate the final system call set, composed of 288 features. More specifically, the x86-arch system calls set is limited to 212 features while the ARM-arch set encompasses the whole 288 feature set.

- **Total System Calls** - total number of system calls issued by the application during the run-time.

As a result of the data processing stage, 489 features were generated to characterize each application that composes each of the *final* device-related datasets. More specifically, 313 numeric, 172 cat- egorical, and 4 list features are provided for each sample in both datasets (i.e., emulator and real device). Regarding the feature type, 289 are *dynamic features*, collected using *advanced* dynamic analysis procedures, while 200 are considered static features, collected using *basic* and *advanced* static analysis techniques.

5. Results

5.1. *Final Datasets: Emulator vs. Real device*

The dynamic step acted as a filter step for the applications, just selecting for each final dataset the ones that were successfully installed and executed on each device. As a result, even though the initial dataset was formed by the same number of apps (i.e., 93,894), split into 58.4% malware apps (i.e., 54,834 samples) and 41.6% benign apps (i.e., 39,060 samples), the *final* datasets show different compositions, as provided in Table 8 and Table 9. The *Initial* column is the same for both tables, as it provides the composition of the *initial* dataset. The column *Install Failed* refers to the applications that were not successfully installed on the device. The most common reason for that was that the app was not compatible, not having the native libraries required for the specific architecture. That error was reported, when trying to install, as *INSTALL FAILED NO MATCHING ABIS*.

Another reason was the lack of valid or any certificates inside the app, thus *INSTALL PARSE FAILED NO CERTIFICATES* message was raised. Both types of errors ended up with the same result, the app was not successfully installed, being discarded, and not further processed for that specific device. The column *Dynamic Analysis* is split into two subcategories: *Failed* and *Processed*. The subcategory *Failed* refers to applications that were successfully installed but due to either of the *monkey* tool not being able to start the application (i.e., main activity not defined) or the debugger failing to attach to the process, the dynamic data was not possible to collect. As a result, those applications were discarded for the next steps. The subcategory *Processed* refers to the apps that were successfully installed and dynamic data were able to be collected. The column *Final*, provides the *final* dataset compositions when duplicated samples (i.e., SHA-256 collisions) are removed from the *processed* data. Therefore, the *final* datasets for each device are the applications from the *initial* dataset that were successfully *installed* and *processed*, thus the *dynamic analysis* was successful, and their hash value was unique. Finally, the rows in these tables indicate the app *class*, malware or benign, and the total values.

The emulator final dataset composition is provided in Table 8. A total of 63,991 apps compose this dataset, indicating that 68,2% of the apps from the initial data were successfully processed and found unique (i.e., 72,5% before the removal of the duplicates). About 4,3% were installed but failed and 23,2% directly failed to be installed. The class composition of this final dataset is slightly unbalanced towards the benign class, being 44.9% malware and 55.1% benign applications. Even though this proportion may seem reasonable, the initial malware dataset was larger than the benign dataset, meaning that a considerable amount of malware applications failed to be installed in the emulator. More specifically, 19,788 malware apps directly failed to be installed, which represents 36.1% of the initial malware dataset versus just 5.1% of the benign samples with respect to the initial legitimate dataset. As a result, 91% of the failed-to-be-installed samples are malware apps. This fact enables to conclude that malware tends to not be compatible with emulators (i.e., x86 architecture), thus being highly constricted by the architecture of the device. Therefore, in the case of malware, incompatibility issues can hamper significantly the collection of dynamic features in emulated environments.

The real device final dataset composition is provided in Table 9. A total of 78,137 apps compose this dataset, indicating that 83,2% of the apps of the initial data were successfully processed and found unique (i.e., 88% before the removal of the duplicates). About 4,6% were installed but failed to be processed and 7,4% directly failed to be installed. The *class* composition of the final dataset is slightly unbalanced towards the malware apps, with proportions closer to the initial dataset. In this regard, 53% of the final dataset are malware apps (58,4% in the original). The 12% of the initial malware dataset failed to be installed on the real device and just 1.1% of the benign applications. Furthermore, and in a similar fashion as in the emulated device, 94% of the failed apps belong to the malware dataset. These figures show that, as in the emulator, acquiring

Table 8 – Emulator final dataset.

Class	Initial	Install Failed	Dynamic Analysis		Final
			Failed	Processed	
Mal	54,834	19,788	2,213	32,822	28,745
Leg	39,060	2,000	1,797	35,263	35,246
Total	93,894	21,788	4,010	68,085	63,991

Table 9 – Real Device final dataset.

Class	Initial	Install Failed	Dynamic Analysis		Final
			Failed	Processed	
Mal	54,834	6,527	2,463	45,844	41,382
Leg	39,060	422	1,860	36,778	36,755
Total	93,894	6,949	4,323	82,622	78,137

dynamic data is more challenging for malware samples than benign applications.

As can be inferred from Table 8 and Table 9, the *duplication* issue was more significant in the *malware* sets than *benign* sets, an expected side-effect of the usage of *overlapping* malware repositories and the scarcity of data sources.

As a result, the comparative inspection of the two final datasets shows that dynamic analysis of malware apps in the emulator is a more challenging task than in the real device. In both malware datasets, similar numbers and proportions of apps were installed but failed to be processed, thus the main difference lies in the *incompatibility issues*, when the apps could not be installed. Although emulators are easier to manage and deploy than real devices, their architecture, inherited from the *host* machine, poses challenges to dynamic malware analysis, as many apps might not be compatible with x86 devices. This fact could condition, impact, and bias the results obtained using emulators. Consequently, the underlying CPU architecture appears to be an important conditional variable to be taken into account when performing dynamic malware analysis.

5.2. *Final Datasets: Soft vs. Hard label*

The final datasets, discussed in Section 5.1, base their composition according to the unique and successfully processed samples by data source, reported in Table 5 and Table 6. Each sample processed was labeled, as either benign or malware, according to the original source of the data (i.e., benign or malware repository), without any further analysis, defining their *source* label or *soft* label.

Nevertheless, after the data processing step, the analysis of the detection results arouse the possibility of *misclassification* issues from the original sources. More specifically, based on the detection reports collected, some allegedly benign samples, gathered from trusted benign sources, reported a non-zero malware detection ratio (i.e., they were detected as malware for at least one AV). Similarly, a small number of alleged malware samples, from well-known malware repositories, were found to have zero malware detection ratios, meaning that none of the AV detected them as malware. As a result, the suspicion of some data *misclassification*, from some sources, shaped the concept of determining a *hard label*. The *hard label* is based on the detection results and strictly applies the following rules:

- A *benign sample* has a zero-valued malware detection ratio AND originates from a trusted legitimate source. The coded *hard* label for benign data is 0.
- A *malware sample* has a non-zero malware detection AND belongs to a malware repository AND a malware family name is reported. The coded *hard* label for malware data is 1.
- Any mismatch with the two conditions stated above is labeled as -1, *indefinite* class. It indicates that the ground-truth label cannot be ensured and further inspection must be performed. Assigning any other label would be prone to *misclassification* issues.

The *hard label* aims to provide better quality data by cumulative evidence towards the label. So, in order to categorize a legitimate sample with the *hard label*, the *soft label* must be supported by the additional evidence of not being detected as malware by any scanner. This does not provide complete certainty about the label, as malware scanners can be bypassed (Cai and Yap, 2016) and have been evaded (Zheng and Xu, 2015), but it creates additional support, *reinforcing* the assigned label. Similarly, the malware *soft label* must be supported by being found as malware for at least one AV and, additionally, having a malware family name reported. This last extra-condition aims to avoid cases where just a few scanners detect the sample as malware in a vague and non-defined way, making it prone to be a *false positive*. Therefore, the *hard label* aims to provide more certainty around the assigned label, reducing the impact of noisy, *misclassified* and low-quality data coming from dubious samples, an essential requirement to build effective machine learning detection systems.

Table 10 – Distribution of labels in the emulator dataset.

Class	Soft label	Hard label	
		Indefinite	Definite
Mal	28,745	91	28,654
Leg	35,246	4,437	30,809
Total	63,991	4,528	59,463

Table 11 – Distribution of labels in the real device dataset.

Class	Soft label	Hard label	
		Indefinite	Definite
Mal	41,382	165	41,217
Leg	36,755	4,856	31,899
Total	78,137	5,021	73,116

The composition of the final datasets regarding the *soft labels* and *hard labels* are provided in Table 10 for the emulator and Table 11 for the real device. The *Indefinite* column corresponds to the dubious samples, categorized as -1 in the *hard label* but assigned with the corresponding 0 or 1 value, depending on the data source in the case of the *soft label*.

As reported in Table 10 for the emulator dataset, 91 samples from the malware dataset are categorized as *indefinite* while 4,437 from the benign dataset, a remarkable difference between both data sources. Nevertheless, from the 4,437 samples of the benign dataset, 2,362 were detected by just 1 AV, 656 by just 2 AVs, and 225 by 3 AVs. As a result, 3,243 of these 4,437 samples are detected by 3 AVs or less, which is far from the average number of positive AV detections for the malware dataset (i.e., 30), which suggests they are likely *false positives*. A total 744 of the 4,437 samples have more than 10 positive detections and just 13 of these are equal or over the average number of positive detection on the malware dataset (i.e., 30 or more AV detections). These facts strongly suggest that the majority of the *misclassified* samples may correspond to *false positives* or *false alarms* triggered by the antivirus scanners.

Similarly, based on Table 11, for the real device dataset, 165 malware samples are labeled as *indefinite* and 4,856 from the benign dataset. As in the emulator case, there is a significant

difference between both datasets. More specifically, from the 4,856 *indefinite* benign samples, 2,695 got just 1 positive AV detection, 680 got 2 and 244 just 3 AV detections. Therefore, 3,619 of the 4,856 samples were detected by 3 AVs or less, far from the mean of positive AV detections for the malware dataset (i.e., 30,47 AVs), which, again, suggests they could be *false positives*. Just 768 of the 4,856 samples are detected by more than 10 AVs and just 12 of them are over the mean of positive AVs detection on the corresponding malware dataset (i.e., 31 or more AV detections). Thus, these figures suggest that, as in the emulator case, the majority of the *misclassified* samples correspond to *false positives*, flagged erroneously by some of the AV scanners.

As a consequence of the *hard labeling* process, removing the dubious samples to increase the quality of the data, the *final* datasets were reduced around 7%. Even though most of the data defined as *indefinite* might be considered *false positives*, only a deeper inspection could guarantee it. As most of the detection mechanisms used by AVs just use static data, they can be easily bypassed but also prone to *false alarms* (Cai and Yap, 2016). Furthermore, as new malware samples are discovered on a daily basis, becoming *known*, these static features are constantly updated. Consequently, samples considered malware-free today might be detected as malware in the future, thus creating uncertainty around the ground-truth label. Therefore, for the sake of rigorousness, dubious samples were categorized as *indefinite*, but it is up to the user of the dataset to assess and decide what samples are included or excluded from the final dataset and what labels are preferred, either the *soft labels*, trusting the source, or the *hard labels*, trusting the AVs results.

5.3. *(When) Time Matters: Metadata Analysis*

The importance of assigning a date to a malware sample arises from the fact of a phenomenon called *concept drift*, directly related to machine learning and forensic analysis. Malware is a non-stationary, constantly evolving phenomenon, with its key features and characteristics prone to change over time. Consequently, relevant features to characterize and discriminate *old* samples may differ significantly from the ones relevant for recent or future samples (Jordaney et al., 2017). As the main consequence of this rapid evolution, it becomes an extremely challenging task to generate models that generalize well over extended periods of time (i.e., detecting well past, present and future samples). *Concept drift* on data severely impacts the performance of ML models over time until they become obsolete. Android malware classifiers and detection rules can become unsustainable in the long term, outdated as malware evolves. Hence it becomes critically important to place malware in its temporal context in order to detect and react against the changes in malware over time and on time (Hu et al., 2017).

Unfortunately, there is no straightforward and reliable technique to date an Android app. As timestamps can easily be modified, purposely or not, the date of Android applications is a challenging, almost infeasible, task. Even though the ground-truth timestamp is unlikely to be achieved, approximations can be deemed useful in this matter. In this research, 4 timestamps are used to date applications in an approximate manner when the ground-truth is not accessible. Therefore, *Earliest Modification*, *Last Modification*, *First Seen VT* and *First Seen ITW* were extracted, providing different approaches to *approximate* the date of a sample.

Table 12 – Timestamps *validity* in the emulator dataset.

Class	Timestamp	Valid	Not Valid
Malware	Earliest Mod	26,985	1,760
	Last Mod	28,395	350
	First Seen VT	28,745	0
	First Seen ITW	6,145	22,600
Benign	Earliest Mod	29,969	5,277
	Last Mod	34,335	911
	First Seen VT	35,246	0
	First Seen ITW	209	35,037

Table 13 – Timestamps *validity* in the real device dataset.

Class	Timestamp	Valid	Not Valid
Malware	Earliest Mod	37,969	3,413
	Last Mod	40,934	448
	First Seen VT	41,382	0
	First Seen ITW	8,090	33,292
Benign	Earliest Mod	31,207	5,548
	Last Mod	35,782	973
	First Seen VT	36,755	0
	First Seen ITW	219	36,536

Table 12 provides information about the *valid* and *not valid* timestamps for each approach for the emulator dataset while Table 13 reports the same information for the real device dataset. For the purpose of this research, a *valid* timestamp is defined as those timestamps that are possible for the app to belong to, disregarding its accuracy, thus encompassing from 2008, the first Android OS version release, to 2020. Contrarily, a *not valid* timestamp occurs when the timestamp data is missing/not defined or the date is not factually possible for the Android app to exist (i.e., dates before 2008 and after 2020).

As can be observed on Table 12 and Table 13, both datasets show similar distribution trends regarding the timestamps. More specifically, in both cases, the *First Seen VT* timestamp shows *valid* values for all the applications in the dataset. Despite its dubious accuracy, as will be explained later, it provides *valid* data for the whole dataset. Contrarily, *First Seen ITW* timestamps are missing for almost all samples in the legitimate datasets and just *valid* for some malware samples. Despite that, they are presumably the most accurate of all the timestamps. Both *First Seen VT* and *First Seen ITW* values, when not missing, they are always comprised between 2008 to 2020, thus being always *valid*.

Regarding the *apk* inner timestamps, the *Earliest Modification* is *valid* for the vast majority of applications but it's certainly not as accurate as the *Last Modification* timestamp, as it tends to refer to an early timestamp, not indicating the *last* app update or release, what *Last Modification* may reflect. The *Last Modification* timestamp provides *valid* information for almost all the samples and it is preferred due to its more likely accurate nature. Finally, as reflected in the provided tables, both for emulator and real device, even though it may seem the op-

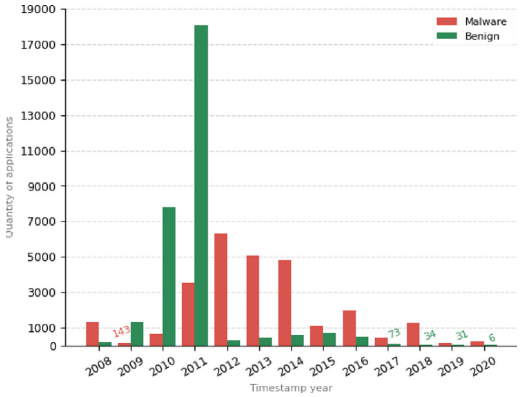


Fig. 2 – Emulator *Earliest Mod valid* year distribution.

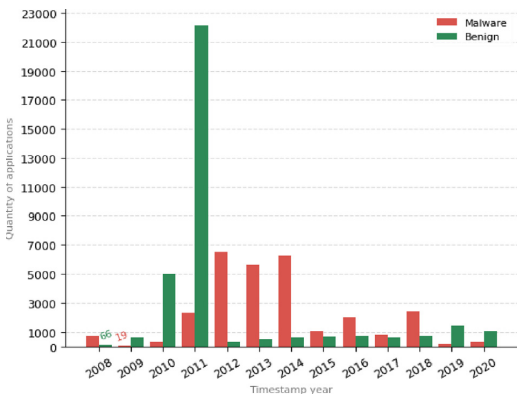


Fig. 3 – Emulator *Last Mod valid* year distribution.

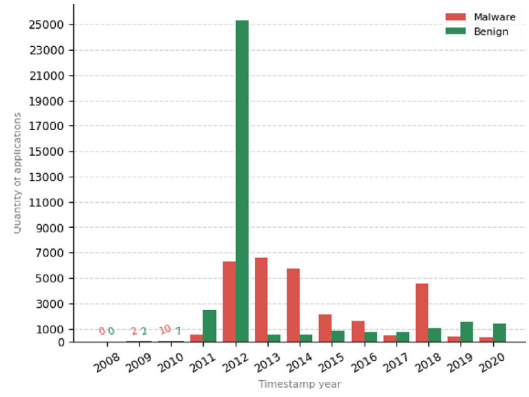


Fig. 4 – Emulator *First Seen VT valid* year distribution.

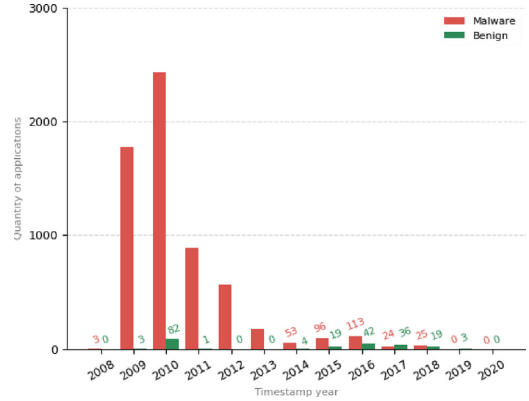


Fig. 5 – Emulator *First Seen ITW valid* year distribution.

posite, legitimate applications have more *not valid* values than malware apps.

As a result, if not altered purposely, the *Last Modification* timestamp should provide accurate infor- mation about an app's creation/release date and it is widely available for most samples, thus being the preferred option. The *First Seen ITW* would also provide accurate temporal context but the lack of values in the vast majority of samples makes it a not preferred choice when a large number of samples are needed. Thus *First Seen VT* arises as the second preferred option due to its complete *validity*, despite its inaccurate, delayed nature, related to the users' activity (i.e., submission time). The *Earliest Modification* seems to provide inaccurate data, not preferred for temporal placement purposes if the information is *not valid* or differs excessively from the *Last Modification* date.

The *valid* distributions for the emulator data set for each of the timestamps are provided in Fig. 2, Fig. 3, Figs. 4 and 5. The green bars represent the benign samples while the red bars correspond to malware apps. For the sake of interpretation, numbers are added for those bars whose frequency is smaller than 150.

The *Earliest Modification* timestamps for the emulator data, provided in Fig. 2 show that the vast majority of *valid* samples are concentrated between 2010 and 2011, while the rest of years show a much smaller proportion of samples. The most recent years have a significantly lower quantity of applications, especially in the case of legitimate applications, thus providing limited data for the years from 2017 to 2020.

The *Last Modification* timestamps, which are provided in Fig. 3, show a similar trend than the *Earliest Modification* timestamp, with most of the *valid* samples concentrated between 2010 and 2011. However, the *Last Mod* data is more spread and populated in recent years, having more than 150 apps per year in all years in the period encompassing from 2017 to 2020. In this case, the very early years show more missing data, with values less than 150 samples per year in 2008 and 2009.

The *First Seen VT* timestamp was found as the most complete data in terms of *validity*. Nevertheless, in terms of *coverage*, as shown in Fig. 4, the data are scarce for the early years, 2008-2010, but more prevalent in the last years. This fact emphasizes that these data are generally subject to *delay* with re-

spect to the *Last Modification* timestamp, thus suggesting lesser accuracy, being based on the users' proactive behavior submitting apps to generate the timestamp.

The *First Seen ITW* timestamp was suggested to be the most accurate (i.e., it marks its first seen online, anywhere) but clearly incomplete, lacking coverage and prevalence in all years as evidenced by the values and skewed distribution in Fig. 5. In this case, information for benign data is significantly missingin all years.

The same trends on the emulator data are confirmed for the real device data. For the sake of readability of the text and its comprehension, avoiding unnecessary repetitions of figures, the real device distributions are provided in Appendix A. As can be observed, in both cases, there is no exact and completely certain way to assign a date to an application. All timestamps analyzed may provide some temporal approximations but the exact date of inception of an application appears to be hardly reachable. The ground-truth timestamp for Android apps seems to be certainly not achievable for any of them.

Last modification, a reference of the last time any of the files inside the *apk* was modified seems the most natural option to assign a date for an application and should provide relatively accurate results. However, this metadata information can be tampered with and modified purposely by attackers. As a result, even though this is not a common practice found in Android malware authors, this fact must be taken into account, not relying completely on any of the approaches as ground-truth but as approximations (e.g., timestamps with last modification in 2107 can be found in the dataset).

First Seen VT is a timestamp based on VirusTotal which informs about when the application was first submitted. This data cannot be tampered with by attackers but it can be inaccurate as depends on the action of users (i.e., someone has to submit the file), thus it can never be exact but relatively delayed, except in the case that the malware author uploaded it to self-check the detection ratio.

First Seen ITW is the time reference that indicates when the application was seen online, anywhere, for the first time. As a result, this timestamp may also be prone to delays, but provide a more accurate timestamp than the *First Seen VT*. It is a better approach than *First Seen VT*, but it is harder to collect and not possible in many cases. *Earliest modification* is placed as the least preferred option for its inaccurate nature as any of the files inside the app could exist much time before the application (i.e., that would be the reported timestamp) or report a not possible value, such as 1980. Therefore, *Last modification* and *First Seen VT* approaches seem the preferred options when the temporal context for an app is needed. Even though their accuracy might be dubious, they can provide a good approximation if they have not been tampered with or not delayed significantly.

5.4. *Malware Family Attribution*

Malware use a great variety of techniques and means to accomplish their malicious intentions. These intentions are embodied in the app's source code by the malware author. Once new malware is detected, the samples are studied by specialists to determine, based on its characteristics, if it is a *known* malware variant or an unknown one. Malware variants are build using the same base code from another known malware (i.e., AVs have developed a signature to detect it) thus composing a malware family. Therefore, a malware family is a set of malware that has been generated from the same source code. When these variants include different tools or techniques they are denoted as descendants (Cohen and Walkowski, 2019).

Malware family attribution allows classifying malware samples into well-known categories, enhancing malware identification, characterization, and detection. However, even though some malware families have well-established denominations, it does not exist any convention on malware family naming. For example, the sample with SHA-256 hash value 727433bf595c257b029812106142d7ae29682a9a125c8038314cc254fd2cdbdf is detected as belonging to the *Steek* malware family for some AV vendors, to *Fatakr* malware family for some others, an unnamed type of Android trojan (i.e., TrojanClicker.AndroidOS.t), just a trojan (i.e., Trojan (0048d7e51)), a generic Android malware (i.e., Android/Generic.AP.8CFF0!tr), malware named using custom cryptic denominations (i.e., Artemis!A879EF0F3DAA), just malware (i.e., Malware (ai Score=100)) or a malware agent (i.e, Andr.Malware.Agent-1518794). Therefore, a single malware sample can receive as many denominations as the number of AV scanners processing the file, even for well-known malware families such as the *Steek/Fatakr* family. As a consequence of the lack of harmonization and convention, malware scanners tend to use their own naming conventions (Kaspersky 2021; Microsoft 2021), sometimes cryptic denominations, which pose challenges to malware family identification and categorization, even for malware analysts (Hahn, 2019). Malware detection is critical to prevention, but proper identification is crucial for malware elimination, cleaning, and restoration after an infection (Hahn, 2019). For research purposes, malware family attribution helps to characterize and understand different malware categories, their evolution, and the development of more specific and effective counter-measures such as for instance, for ransomware.

Aiming to minimize the malware family naming confusion, all *final* datasets samples were scanned using VirusTotal AV engine (VirusTotal 2020). From the detection reports obtained, the most prevalent malware family provided by the scanners was imputed to the specific sample (i.e., the majority of the vote among all positive scanner results). The heuristic procedure was executed as follows:

1 An initial database of malware families was generated from online resources and malware families research studies. When possible, malware families with several denominations were categorized in a single label, which included all denominations separated by a slash ("/"). For instance, *"Steek/Fatakr"*.
2 All samples, including the allegedly benign ones, were scanned, one at a time, using the AV scanner engine, and a detection report was retrieved for each app.
3 The app's detection report was parsed and all the positive scanner results were confronted with the known malware family names in the database.

Table 14 – Top-15 malware families in final data sets.

Emulator			Real Device		
Family	Total	%	Family	Total	%
Airpush	6,521	27	Airpush	7,775	22
Boxer	3,557	15	SMSreg	5,019	15
Malap	2,574	11	Malap	4,055	12
FakeInst	2,158	9	Boxer	3,597	10
Agent	1,837	8	Agent	2,934	9
SLocker	1,822	8	FakeInst	2,384	7
BankBot	1,241	5	SLocker	1,846	5
FakeApp	1,064	4	BankBot	1,297	4
Dowgin	772	3	Dowgin	1,145	3
GinMaster	595	2	FakeApp	994	3
Kuguo	513	2	DroidKungFu	990	3
SMSreg	497	2	Kuguo	843	2
Youmi	447	2	GinMaster	827	2
DroidKungFu	269	1	Youmi	628	2
Simhosy	232	1	Simhosy	399	1
Total	24,099	100	Total	34,733	100

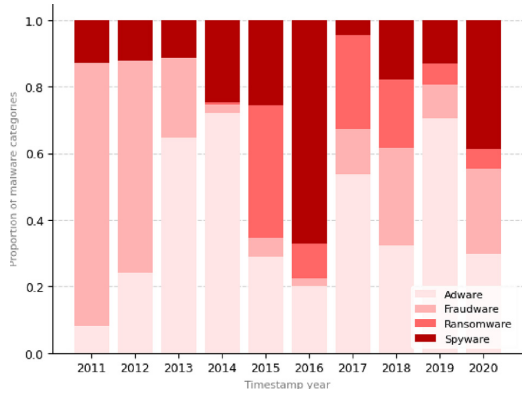


Fig. 6 – Emulator categories distribution along years using *First Seen VT* timestamp.

4 A family name from the database was imputed to every sample based on the majority of the vote of all the individual positive results. In case of positive detection but no malware family was imputed (i.e., the malware family was unknown or not included in the database), the report was analyzed manually and imputed by manual inspection if a malware family was reported. The new suggested malware family was added to the database and the majority of the voting process was repeated.

As a result of these heuristics, 99.7% of the samples present in the *final* datasets have a malware family attribution. More precisely, 209 malware families are represented in the emulator dataset while 240 in the real device dataset. Therefore 31 malware families are represented in the real device dataset but not in the emulator dataset. Even though this difference is consequent with the smaller set of samples that compose the final malware set on the emulator, as discussed in Section 5.1, it also evidences that some malware families are specifically tailored for ARM devices, thus not compatible with x86 emulators. Therefore, in these cases, the applicability of emulator sandboxes in the study and detection of such families is not possible, limiting the capabilities of emulators as forensics and detection platforms.

Table 14 shows the 15 most prevalent malware families in each dataset, ranked in descending order. The top 15 malware families account for a total quantity of 24,099 samples in the emulator dataset (i.e., 84.1% of the *hard label* malware set) and 34,733 in the real device dataset (i.e., 84.26% of the *hard label* malware set).

As reported in Table 14, the same malware families compose the most prevalent 15 malware families in both datasets. However, even though the total proportion of these families on both datasets is almost identical (i.e., ˜ 84%), the distribution differs significantly for some malware families. *Airpush* is the most prevalent family on both datasets while the second most prevalent family differs notably on both datasets. *SMSreg*, which is the second most prevalent family in the real device, with 5,019 samples, representing approximately 15% of the top 15 samples, places very low in the emulator dataset, as this family is just represented by 497 samples, the 2% of the top 15. This remarkable difference indicates that *SMSreg* installation may require, in many variants, ARM native libraries to collect device and user-related sensitive information that is usually not available or ill-defined on emulators, such as the International Mobile Equipment Identity (IMEI) number (F-Secure 2021). A similar but less significant difference is found on *Airpush* and *Boxer* families. The rest show similar proportions on both datasets.

The top 15 malware families can be embedded into four major malware categories (i.e., adware, fraudware, spyware, and ransomware). The color intensity in Table 14 groups them and indicates the degree of threat they pose for the users (i.e., darker meaning riskier). Adware trojans, referenced with the lightest red color, are represented by 6 malware families, being Airpush the most aggressive and prevalent (i.e., Airpush, Agent, FakeApp, Kuguo, Dowgin and Youmi). Besides, 3 fraudware trojans (i.e., Boxer, FakeInst and SMSreg), the first ransomware for Android (i.e., Slocker) and 5 spyware apps (i.e., DroidKungFu, GinMaster, BankBot, Simhosy and Malap) complete the list. Even though they all pose security threats and high risk for Android users, those using exploits and leveraging root privileges, with the purposes of extorsion (i.e., ransomware), persistent access, or stealing sensitive data (i.e., spyware) are of special concern.

The four malware major categories and their distribution along years in the datasets are depicted in Figs. 6 and 7. The timestamp used is *First Seen VT*, indicative of the year when *VirusTotal* received the sample for the first time. This temporal context gives an indicator of malware evolution and trends over time. More concretely, Fig. 6 shows the proportion of each malware category from the most prevalent families along years for the emulator dataset. Fig. 7 shows the same information for the real device dataset. As can be noticed, both graphs show similar trends, indicating a ransomware outbreak in 2015. Adware and fraudware were the most prevalent families in the early years of Android while more recently, with the surge of smartphones, spyware has become more prevalent. According to the data collected, 2017 is a partic-

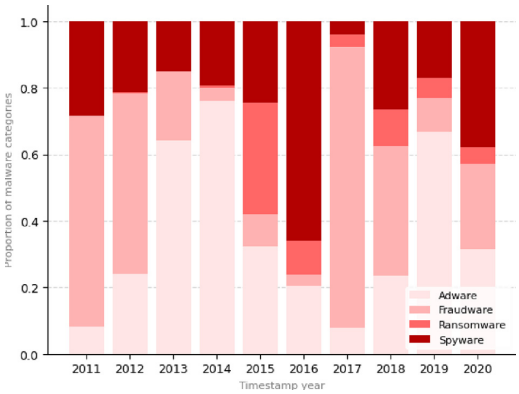


Fig. 7 – Real device categories distribution along years using *First Seen VT* timestamp.

Fig. 8 – Emulator categories distribution along years using *Last Modification* timestamp.

ular year where an increase in adware, fraudware, and ransomware disrupted the increasing trend of spyware.

The *First Seen VT* timestamp is a good indicator of the general trend, as can't be manipulated by attackers, but is limited to the *VirusTotal* activity time (i.e., not much data before 2011) and dependent on the behavior of the users, in relation to when the file is submitted to scan. A different and complementary perspective can be obtained using the *Last Modification* timestamp. When not tampered with, it allows getting a more exact timestamp about the application's temporal context Figs. 8 and 9 provide the proportion for the top 15 families major categories along years, using the *Last Modification* timestamp.

As can be noticed from Fig. 8 and Fig. 9, there exists a similar trend as shown by the *First Seen VT* timestamp but with the remarkable exception of ransomware apps. In both cases, the outbreak of ransomware apps has its starting point in 2014, thus noticed by *VirusTotal* with delay. This fact is confirmed in Lipovsky´et al., 2016, placing 2014 as the year

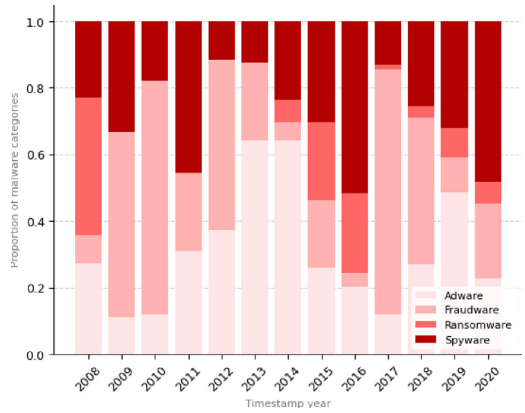


Fig. 9 – Real device categories distribution along years using *Last Modification* timestamp.

when ransomware first appeared in Android devices. However, using this timestamp, 2008 shows a significant prevalence of ransomware which is not possible. This fact suggests that the timestamps are tampered with for some specific ransomware apps, misplacing these data in the timeline. Despite this fact, the general trend and proportions are confirmed using both temporal contexts, evidencing the goodness of both approaches as approximate timestamps to date applications and trends in malware families.

5.5. *Dynamic App Profiling*

The dynamic features collected for all the applications in both datasets are system calls, also referred to as *kernel* calls or *syscalls* for short, issued at run-time without any user interaction (i.e., for max. 60 seconds). The total feature set of system calls is composed of 288 Android OS system calls. For each sample, the absolute frequency (i.e., total number) of each system call during the run-time is reported. An additional feature, *total syscalls*, was constructed, reporting the summation of the total number of syscalls issued by the application during its execution. As explained before, even though different architectures have different system calls set, for the sake of homogeneity of the datasets' features, the larger system call set (i.e., real device) was used to define the total amount of system call features (i.e., 288). For that reason, more 0-valued or not-issued system calls can be found in the emulator dataset than in the real device dataset.

The Table 15 provides a summary of descriptive statistics computed for the *total syscalls* feature in both datasets, for each app class (i.e., benign and malware) and for both labels (i.e., soft and hard). To improve the interpretability of the table, the *soft labels* statistics are provided in white-colored cells while the dark-colored cells provide the information regarding the *hard labels*. The statistics calculated are: mean, standard deviation (referred as S.D.), range interval (i.e., minimum and maximum value), quartiles (i.e., 25% - Q1, 50% - median, 75% - Q3) and interquartile range (i.e., $IQR = Q1 - Q3$). The total amount of samples of each dataset is reported as the n value.

Table 15 – Descriptive statistics of system calls.

Device	Class	Label	Statistic							
			Mean	Median	Range	25%	75%	IQR	S.D.	n
Emulator	Malware	*Soft*	7,711	2,363	[0, 1,144,643]	765	5,894	5,129	27,893	28,745
		Hard	7,694	2,361		763	5,888	5,125	27,911	28,654
	Benign	*Soft*	8,755	2,959	[0, 705,335]	1,136	6,585	5,449	20,995	35,246
		Hard	8,664	2,890		1,127	6,591	5,464	20,771	30,809
Real Device	Malware	*Soft*	10,390	3,258	[0, 524,761]	963	8,895	7,932	26,232	41,382
		Hard	10,410	3,267		962	8,916	7,954	26,272	41,217
	Benign	*Soft*	2,878	1,148	[0 - 533,765]	478	2,102	1,624	13,386	36,755
		Hard	2,792	1,134	[0 - 529,279]	473	2,090	1,617	12,860	31,898

The average and median provide information about the central tendency values of the *total syscalls* distribution for each dataset, type, and label. They provide the notion of *middle* or *expected* values while the rest of statistics are related to *dispersion* or *variability* of the data.

As can be noticed, even though in the *benign* datasets case, the *soft* and *hard* labels datasets sizes (i.e. *n* value) are significantly reduced, the values of the statistics are not really affected, showing similar values, slightly reduced for almost all statistics in the case of *hard* labels. In all cases, when malware and benign datasets are observed, the mean is significantly greater than the median, which is not greater than 3,300 in any case. This fact shows that 50% of the apps, either malware or benign, issued less than 3,300 syscalls and that there are some *extreme values* or applications that issued a huge amount of syscalls (i.e., outliers). Furthermore, 75% of the apps issued less than 9,000 syscalls with the central 50% of apps grouped around a range of maximum 7,954 syscalls, which emphasizes the fact that there is a great majority of applications issuing not more than 9,000 syscalls and a minority of applications issuing much greater values, as evidenced by the maximum values in all ranges, over 524,000 syscalls. This fact proves that the *total syscalls* distribution is not symmetrical in any case, being right-skewed with some outliers that affect the value of the mean, making it a not good representative of the *expected* value.

The computed statistics reported in Table 15 show remarkable behavioral differences between emulators and real devices. The statistics considerably differ between classes on both datasets. In the case of malware, the statistics show that malware apps are less *active* in emulators than in real devices, with just a few extreme exceptions that have a powerful impact on the mean. This fact could suggest that some *sophisticated* malware may detect the emulator environment thus becoming less active and hide its real behavior. However, these differences could also be originated by the significant difference between both datasets, suggesting that ARM-dependent apps may be the cause of such increase. Further investigation should be performed to analyze if the cause of these behavioral differences is caused by the platform or the app. On the contrary, in the case of benign apps, with similar dataset sizes, the statistics show a significant more *active* behavior in the emulator than in the real device. More specifically, the *most* active apps (i.e., over the third quartile) issued more than 6,600 syscalls in the emulator while in the real device this value is slightly over 2,000. These behavioral differences were already pointed out by Guerra-Manzanares et al., 2019, suggesting that an app *behavior* (i.e., system calls) might not be *identical* across devices. This fact challenges the identical behavior assumption usually made in research studies to justify any platform selection. A more detailed inspection of these differences at the system call level is provided in Table 16.

The total quantity of distinct system calls actually used per dataset (i.e., used syscalls set, at least once), with respect to the system calls feature set (i.e., 288) are reported in the fourth and fifth column of Table 16. The subset of the five most used syscalls and the proportion of apps in the dataset that *issued* them, at least once, are reported in columns sixth and seventh. The eighth column reports the *most issued syscalls* with respect to the *total amount of issued system calls* by all the apps in that specific dataset, a figure reported in the tenth column. The ninth column provides information about the proportion of syscalls over the total quantity of syscalls issued on that specific dataset that was *caused* by each of them. Similarly as before, the *soft* and *hard* labels datasets show similar values and sets. They are reported for the sake of completeness, but their minor differences are not further discussed.

As reported in Table 16, not all the *available* syscalls were issued by the applications at run-time. The variability of the syscalls is reduced to a smaller subset of them, issuing, at most 44.4% of the whole syscalls set (i.e., 128 out of 288 syscalls). This fact seems even more remarkable in the emulator malware dataset where the smallest subset of syscalls is found, as just combinations of 99 syscalls are issued by 28,745 apps. However, it is worth noting that even it may look a considerably smaller fraction, the final syscalls feature set was related to the ARM architecture, which showed more syscalls *available* than the x86 architecture thus the proportion is greater when taking into account its *own* feature set. As referenced before, the larger ARM-architecture syscalls set was used for the sake of homogeneity of the datasets as it includes the x86-related ones. Thus, when taking into account just the specific x86 syscalls (i.e., 212), the fraction for the emulator grows to 46.7% for the malware dataset and 57.5% for the benign dataset. This fact suggests that these differences are due to the extended feature set used, but not that in the emulator fewer system calls are used. However, a real difference is found between the size of the syscalls set used by malware and benign samples. In both platforms, the benign samples use a larger set of syscalls than the malware. Benign samples consistently use a wider variety of syscalls even when the to-

Table 16 – System calls usage statistics.

Device	Class	Label	Used syscalls set	% total syscalls set	Most used syscalls	% total apps	Most issued syscalls	% total syscalls	Total issued syscalls
Emulator	Malware	*Soft*	99	34.4	ioctl getuid32 mmap2 futex close	99.4 97.8 96.8 96.7 96.7 99.3 97.8 96.8 96.7	read getuid32 write epoll pwait ioctl	11.7 11.6 11.1 11.1 11.1 11.7 11.6 11.1 11.1	221,659,298
		Hard	99	34.4		96.7		11.1	220,454,507
	Benign	*Soft*	122	42.4	ioctl getuid32 mmap2 close futex	99.9 99.2 98.9 98.7 98.6 99.9 99.5 99.2 99.2	read write ioctl recvfrom epoll pwait	20.4 13.8 13.2 9.2 7.0 20.3 13.8 13.2 9.3	308,595,229
		Hard	120	41.7		99.2		6.9	266,918,501
Real Device	Malware	*Soft*	113	39.2	clock_gettime getuid32 ioctl futex mmap2	98.9 97.6 97.6 96.0 96.0 98.9 97.6 97.6 96.0	clock_gettime ioctl getuid32 mprotect SYS 329	37.9 11.7 5.7 5.0 4.4 37.9 11.7 5.7 5.0	429,999,343
		Hard	112	38.9		96.0		4.4	429,069,377
	Benign	*Soft*	128	44.4	clock_gettime getuid32 ioctl writev read clock_gettime getuid32 ioctl read	99.3 97.4 97.1 95.6 95.5 99.7 97.7 97.3 96.0	clock_gettime getuid32 ioctl SYS 329 mprotect	35.3 7.3 6.6 6.0 5.9 36.0 6.9 6.7 5.8	105,779,886
		Hard	126	43.8	writev	95.8		5.8	89,051,861

tal amount of syscalls issued is significantly lower, as in the case of the real device.

The *most used syscalls* and *most issued syscalls* sets, reported in the sixth and eighth columns, are similar across class types (i.e., malware and benign) within the same platform but completely different between platforms. More specifically, the syscall *clock_gettime* appears to have a remarkable significance in the real device, causing over 35% of the total amount of syscalls for that device in all cases, and being issued, at least once, by 98.9% of the apps run in that platform. Contrarily, in the case of the emulator, this syscall does not show the same prevalence, being excluded from the top 5 syscalls subset. For the emulator, two distinct syscalls arise as *most used* and *most issued*, *ioctl* and *read*, respectively. The *read* syscall appears to be much less important for the real device dataset than for the emulator dataset. It is the most issued syscall in the emulator, accounting for between 11.7% to 20.4% of the total syscalls, but excluded from the most issued calls set for the real device.

The total issued syscalls, reported in the last column of Table 16, provides additional support to the fact, stated before, that benign apps show a distinct, more active behavior, in the emulator than in the real device. More precisely, even though the benign datasets have similar sizes, the total issued syscalls figure is three times greater in the emulator than in the real device. However, in the case of malware, as the datasets are of remarkably different sizes, the same statement is not applicable. Therefore, Table 16 provides additional support to the distinct behavioral differences found by previous research regarding distinct Android OS platforms with respect to system calls (Guerra-Manzanares et al., 2019; Guerra-Manzanares et al., 2019).

5.6. *Static App Profiling*

Each application in the dataset is characterized by 200 static features. Permissions, the most used static features used for malware detection, account for the largest number of static features (i.e., 173). In this regard, 166 binary features report the *standard* permissions requested by apps (i.e., 1/0 value meaning requested/not requested), 3 features are constructed as category counts (i.e., normal, dangerous and signature), 1 binary *custom* permission indicator (i.e., 1 if custom permissions are defined, 0 otherwise), 2 subtotal permission counts (i.e., *total standard* and *total custom*) and a total count of all permissions requested (i.e., sum of *standard* and *custom* permissions) as reported in Table 7.

Table 17 – Descriptive statistics of permissions.

Device	Class	Perm	Label	Statistic								
				Mode	Mean	Median	Range	25%	75%	IQR	S.D.	n
Emulator	Malware	Std	*Soft*	12	10.4	9	[0, 86]	7	12	5	6.2	28,745
			Hard									28,654
		All	*Soft*	9	11.1	10	[0, 116]	7	14	7	7.7	28,745
			Hard								7.6	28,654
	Benign	Std	*Soft*	2	4.1	3	[0, 69]	2	5	3	4	35,246
			Hard		3.8		[0, 56]	1		4	3.6	30,809
		All	*Soft*	2	4.3	3	[0, 104]	2	6	4	4.5	35,246
			Hard		4.1		[0, 80]	1	5		4.1	30,809
Real Device	Malware	Std	*Soft*	12	13.4	11	[0, 97]	7	16	9	9.1	41,382
			Hard					8		8		41,217
		All	*Soft*	9	15.2	11	[0, 124]	8	17	9	12.4	41,382
			Hard									41,217
	Benign	Std	*Soft*	2	4.2	3	[0, 70]	2	6	4	4.2	36,755
			Hard		3.9		[0, 63]	1	5		3.9	31,898
		All	*Soft*	2	4.4	3	[0, 104]	2	6	4	4.9	36,755
			Hard		4.2		[0, 93]		5	3	4.4	31,898

The *total standard permissions* provides the total quantity of Android standard permissions declared by the apps. The upper bound is set at 166 permissions, the maximum number of *standard* permissions at the present time (Android 2021). Table 17 provides a statistical summary of *standard* permissions (i.e., named as *Std* in the *Perm* column) and all permissions counts (i.e., summation of *standard* and custom, named as *All*) by *device* type, app *class* and *label*.

As can be noticed, in the case of permissions, there is a remarkable difference in all the central tendency measures (i.e., mode, mean and median) between malware and benign apps. In this regard, malware applications are characterized by requesting a larger number of *standard* and *total* permissions than benign applications, in both platforms. When the *hard* label is used, these differences are further emphasized. The measures of dispersion (i.e., range, quartiles, *IQR* and *S.D.*) support the same overall tendency. A remarkable difference exists between the emulator malware and the real device malware datasets. In this regard, the real device dataset, more populated, shows larger mean and median values and larger dispersion, suggesting that the malware that did not run in the emulator provide even more notable differences between legitimate and malware applications. It is worth noticing that some malware applications requested 97 *standard* permissions (i.e, 124 if custom permissions are added) while the maximum number in the benign dataset is 70 (i.e., 104 if custom permissions are added).

The histograms depicted in Fig. 10 illustrate the absolute frequency distributions of the *standard* permissions for each device type and label. The horizontal axes refer to the total quantity of requested *standard* permissions in the closed range [0, 70], as they concentrate the 99th percentile of apps. The vertical axes refer to the absolute frequency of apps (i.e., the quantity of apps) that requested each specific number of permissions.

All the distributions in Fig. 10, as suggested by the values in Table 17, are not symmetric, being right-skewed (i.e., longer right tail than left tail). The peaks of both distributions, which relate to the most frequent value (i.e., mode), are different, being higher and closer to zero in the case of benign samples, in all cases. The variability of the malware data is greater than the benign data, encompassing a wider range of values along the horizontal axis. Furthermore, the relatively small overlapping area between the two distributions confirms the existence of two significantly different distributions and characteristics for both classes regarding requested *standard* permissions.

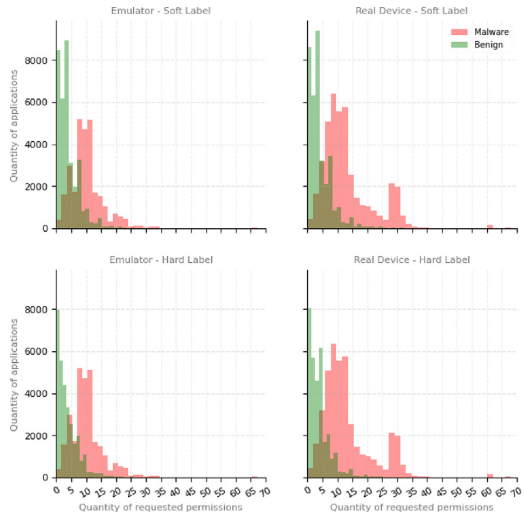


Fig. 10 – Frequency distributions of *standard* permissions.

Table 18 provides information about the most requested *standard* permissions and the *standard* permission set usage for each *device* type, class and label. As the label (i.e., soft or hard) does not change the output significantly, it is not further discussed.

As can be noticed in Table 18, malware apps in both datasets use a smaller fraction of the available *standard* per-

Table 18 – Permission usage statistics.

Device	Class	Label	Used perm set	% total perm set	Most used permissions	Permission category	% total apps
Emulator	Malware	*Soft*	124	74.7	INTERNET	normal	96.8
					READ PHONE STATE	dangerous	88.5
					ACCESS NETWORK STATE	normal	78.9
					WRITE EXTERNAL STORAGE	dangerous	75.0
					ACCESS WIFI STATE	normal	54.1
		Hard	123	74.1			96.8 88.6 78.8 75.0 54.1
	Benign	*Soft*	142	85.5	INTERNET	normal	81.4
					ACCESS NETWORK STATE	normal	50.8
					WRITE EXTERNAL STORAGE	dangerous	35.7 26.5 20.0
					READ PHONE STATE	dangerous	80.0
					ACCESS COARSE LOCATION	dangerous	48.6 34.4 23.7 19.9
		Hard					
Real Device	Malware	*Soft*	129	77.7	INTERNET	normal	97.7
					READ PHONE STATE	dangerous	91.5
					ACCESS NETWORK STATE	normal	84.9 81.0 66.3
					WRITE EXTERNAL STORAGE	dangerous	97.7
					ACCESS WIFI STATE	normal	91.6 84.9 81.0 66.3
		Hard	128	77.1			
	Benign	*Soft*	144	86.7	INTERNET	normal	81.7
					ACCESS NETWORK STATE	normal	51.3
					WRITE EXTERNAL STORAGE	dangerous	36.8 27.4 20.1
					READ PHONE STATE	dangerous	80.2
					ACCESS COARSE LOCATION	dangerous	49.2 35.5 24.4 19.9
		Hard					

missions than the benign sets, even though, as shown in Table 17 and Fig. 10, they tend to request more permissions than benign apps. However, legitimate apps, which, on average, tend to declare a remarkably smaller amount permissions, request a wider variety of them. Columns 6th and 7th in Table 18 show the 5 most used permissions set. As can be observed, 4 of the 5 permissions are common for both classes: *INTERNET*, *READ_PHONE_STATE*, *ACCESS_NETWORK_STATE*, and *WRITE_EXTERNAL_STORAGE*, but with different orderings and significantly different prevalence. The INTERNET permission is the most requested for both classes of applications and across devices. This permission allows the app to open network sockets and is considered a *normal* permission, thus being granted automatically when the app is installed. It is requested by 8 out of 10 benign apps and almost every malware app. After this common first permission, different order of the *common* features appear. READ_PHONE_STATE allows the app to access the *phone state*, which includes the phone number, IMEI, cellular network information, status of any ongoing calls, and a list of the accounts registered on the device. It is considered a *dangerous* permission, being requested by a significantly more fraction of the malware (i.e., 90%) than the benign apps (i.e., 24%). Aligned with the information provided in Table 17, the top four permissions on each class ranked list are requested by roughly 75% of malware apps while decreasing to 25% in the case of benign apps. As a result, in general, based on the most requested permissions, the malware seems to be more interested in accessing *sensitive* data and ensuring connectivity (i.e., 3 of the top 5 permissions are related to networks) while benign apps most used permissions encompass a wider variety of access requests, including GPS location. The reduced list provided in Table 18 seems to imply that benign apps request more *dangerous* permissions than malware apps. In order to explore this fact, datasets' features *normal*, *dangerous* and *signature*, which provide the total amount of permissions requested by the app that lay within those categories, are represented using boxplots in Fig. 11. Boxplots are a visual and condensed way to plot similarly ranged distributions. As there weren't visual nor numerical differences between the boxplots of both labels, in order to avoid repetition, just the *soft* label data is provided. The horizontal axis provides the categories values, namely, normal, dangerous, and signature while the vertical axis provides the number of permissions requested. The body of the boxplot (i.e., solid color fill) is defined by the Q1 and Q3 (i.e., 25 and 75 percentiles, respectively), the central 50% of the data, defining the *IQR* range. The whiskers extend to the so-called *maximum* and *minimum* thresholds, computed as $maximum = Q3 + 1.5*IQR$ and $minimum = Q1 - 1.5*IQR$. They allow categorizing data points that extend further as outliers or *extreme* values (i.e., omitted in this graph). The body of the boxplots is crossed by a solid blue line which indicates the median or 50th percentile. A dotted orange line indicates the mean or *average* value.

The boxplots in Fig. 11 confirm the fact that malware apps use, in general, many more permissions than benign applications, in all three categories. More specifically, the boxplots

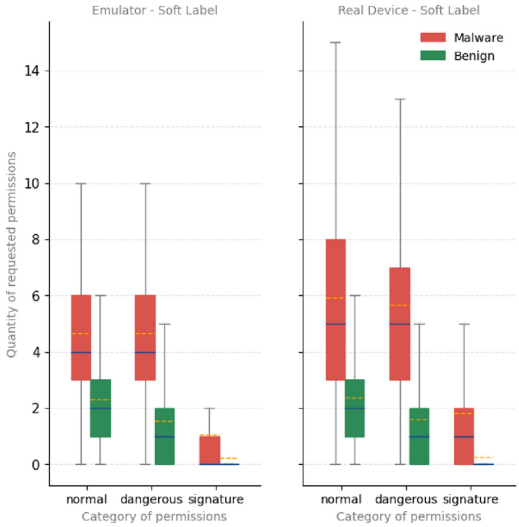


Fig. 11 – *Standard* permissions categories boxplots.

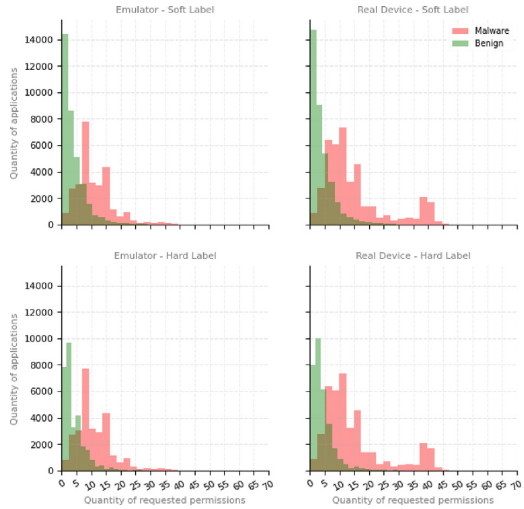


Fig. 12 – Frequency distributions of *all* requested permissions.

show that the lower 25% of malware apps request a similar quantity of permissions than the 75% of benign apps. This fact is specially remarkable in the case of *dangerous* and *signature* permissions. In the case of malware, these differences are further emphasized, with longer bodies and whiskers in all three categories, for the case of real device malware than for the emulator malware. As can be observed, in all boxplots, the mean value is over the median, which suggests the presence of skewed distributions, with extreme values or *outliers* in the right tail of the distributions, as shown in Fig. 10. The median values, better centrality measures in skewed distributions, show that in malware the middle values are in 4 requested permissions for normal and dangerous permissions while for benign they are placed in 2 for normal and 1 for dangerous. Therefore, the boxplots demonstrate the significant differences in the number of permissions in general and categories in particular between malware and benign applications and the tendency of malware to request *over-privileges* from the system.

Another important aspect when dealing with permissions analysis is the presence of *custom* permissions. Although they cannot be traced in an analogous manner as *standard* permissions, as there is no common reference, the presence of them might be considered itself a powerful feature. Two *custom* permissions-related features are generated in this dataset: the presence of them (i.e., binary indicator feature) and the quantity of them (i.e., total *custom* permissions). Table 19 provides a summary of both features for each dataset, class, and label. The column *custom perm declared* provides the proportion of apps that declared at least one *custom* permission and the last 5 columns report a summary of relevant descriptive statistics for those applications that declared *custom* permissions.

As provided in Table 19, 32.1% of malware apps on the emulator declared *custom* permissions, a 2.5 times greater prevalence than the benign apps. Besides, these differences increase even more in the real device, as 42.3% of malware apps show *custom* permissions, thrice more than the benign apps. In the case of the malware class, the *soft* and *hard* labels do not seem to make any difference while in the benign case they show a reduction in the variability, which reduces the mean, range, and standard deviation values. In the real device, where the malware set is larger makes *custom* permissions more prevalent, as shown by greater mean, median, and standard deviation statistics. The range values show that in the most extreme cases even 42 *custom* permissions are defined for malware and 35 for benign data. In any case, the usage of custom permissions appears to be, at least 2.5 times more prevalent for malware apps than benign apps, ranging from 1 in the majority of cases to over 24 *custom* permissions declared in the most extreme cases, for both classes.

Finally, to further analyze the differences in requested permissions between malware and benign apps, the histograms in Fig. 12 depict the absolute frequency distributions of *all* declared permissions for each device type and label. *All* declared permissions sum together the declared *custom* and *standard* permissions. To provide a better comparison with Fig. 10, the horizontal axes are restricted to the closed range [0, 70] of requested permissions. The vertical axes refer to the absolute frequency of apps (i.e., the quantity of apps) that requested each specific amount of permissions. Note that the vertical axes in Fig. 12 reach 14.000, almost twice as in Fig. 10, a fact that may, at first sight, jeopardize the differences when comparing both histograms.

In consequence, as evidenced by the histograms in Fig. 12, when all permissions are considered, the distributions show greater skewness and greater peak differences in contrast to Fig. 10. Further emphasizing the significant differences in permissions sets and quantities between devices, labels, and especially between classes. More precisely, almost all benign applications lay in the range [0, 5] of requested permissions,

Table 19 – *Custom* permission statistics.

Device	Class	Label	*Custom* perm declared	Statistics				
				Mode	Mean	Median	Range	S.D.
Emulator	Malware	*Soft*	32.1%	1	2.2	1	[1, 42]	3.0
		Hard						
	Benign	*Soft*	13.6%	1	1.7	1	[1, 35]	1.5
		Hard	13.2%				[1, 24]	1.2
Real Device	Malware	*Soft*	42.3%	1	4.3	2	[1, 42]	4.7
		Hard						
	Benign	*Soft*	14.0%	1	1.8	1	[1, 35]	1.8
		Hard	13.5%		1.7		[1, 30]	1.4

where just a small fraction of malware is located. The vast majority of malicious apps are spread along the horizontal axis but mainly concentrated in the [5, 25] interval, extended even further in the case of the real device malware data.

6. Discussion

The available datasets used for Android malware detection are mainly focused on the apps collected from single sources covering short time frames, frequently from Android early years. The related literature studies have mostly utilized these limited datasets. This fact neglects the detrimental effects of malware evolution on detection systems, especially the machine learning-based ones. *KronoDroid*, the dataset presented in this research mainly aims to introduce comprehensive malware and benign app samples that encompass the whole lifetime of Android since its initial release until recent years. We incorporated a well-established temporal perspective into the dataset by including various timestamps that can be extracted from the apps and VirusTotal. The inclusion of the *time* variable in Android *apps* is a challenging task for which there is no perfectly certain ground-truth or timestamp that could be used in all cases. Given this difficulty, in our dataset, we aim to present various timing options to the researchers so that they allow setting the seed for novel future Android malware research and enhanced detection systems that could not be performed otherwise. To the best of our knowledge, our dataset is the first one that contains a comprehensive temporal context that can be utilized in the Android malware research.

In this regard, the timestamp of apps allows the investigation and characterization of *concept drift* in Android apps and its proper detection to build more effective and robust models. Although the existing research demonstrates that it is possible to create learning models with high accuracy results, our dataset enables to conduct research that further test the applicability of models in more realistic situations. The research questions regarding the sustainability of high detection capability, for instance, the required time-frame for model updates and their implications on operational and maintenance aspects of malware detection solutions, cannot be addressed without such a dataset.

Machine learning models have promised to revolutionize the intrusion or malware detection domains by identifying new variants of malicious activities where the dominant signature-based approaches mostly fail. If the train and test dataset splitting strategies of usual machine learning workflows are utilized then the models may provide good results on the testing phase, but not generalize well to real *unknown* samples in production and deployment phases. Thus, proper model testing requires a tweak so that the test data should include *unknown* or *future* malware variants which are not contained in the training dataset. Additionally, the detection rates of such samples should be reported. The temporal context of our dataset enables to easily apply such splitting strategy, generate proper validation/testing sets and, thus, enhance the generalization capabilities of the induced ML models in a domain having a frequently changing threat landscape.

Our dataset also provides the family categories of the malware samples. Therefore, the evolution of malware families and their phylogenetic properties can be studied thoroughly when samples are placed in their temporal context. Detection models addressing specific malware families (e.g. ransomware detection) could be developed. In addition to the induction of enhanced detection models, it is possible to do characterization of families, which can lead to more generalized knowledge about mobile malware.

A malware detection solution can work on cloud or device platforms, meaning that the source platform of a train or test sample may vary in a real-life application. In our previous work, we demonstrated that system calls obtained from an emulator or a real device can be different so that such variations may cause reduced detection rates if the source platforms are not taken into consideration during the model lifecycle (Guerra-Manzanares et al., 2019). *KronoDroid* is composed of hybrid datasets collected from an emulator and a real device. Thus, the researchers and practitioners can thoroughly investigate the impact of source platforms on the model outputs and evaluate their detection systems that may work on different system architectures.

KronoDroid provides a wide variety of static and dynamic features to characterize the apps within the datasets. Complementing both approaches with each other provides a more complete characterization that aims to overcome the limitations of single static or dynamic approaches, a common characteristic in the existing datasets and related solutions. We make our dataset publicly available and share it in a ready-to-use structured format. This format might attract more interest not only from the cybersecurity research community but from the machine learning community as well. Although the procedures about data collection from mobile apps are known and applied in various studies, sharing structured data rather

than applications themselves removes the technical difficulty barrier for the research groups not familiar with the cybersecurity test set-ups.

6.1. *Challenges and Limitations*

In the research and data generation process the researchers always face challenges that may have an impact on the collected data or the results. In the case of this research, there are several challenges that may pose limitations on the data and the results obtained.

The collection of dynamic data requires the usage of *live* devices thus requiring the selection or usage of specific operating system versions that may have an impact on the data collected. For this research, Android 8.0 *Oreo* was selected as it was the most *stable* distribution when this research started (i.e., Android 9.0 was the most recent release) and it also provided fewer compatibility issues with old samples that fail to execute in more recent versions of the OS. In addition, when testing with emulators and real devices in analogous settings, the emulator platforms delay the *stable* releases thus limiting the availability of OS selection. Furthermore, the usage of non-recent versions of the OS is a common practice in time-extended studies in order to boost compatibility and analysis quality. For instance, the authors of *AndroCT*, a dataset released in 2021, used Android 6.0 version to execute the testing samples (Wen Li and Cai, 2021).

Another challenge emerges with the usage of emulators as incompatibility issues may arise (i.e., not all the apps support these architectures) or the behavior of malware on them might be distinct (e.g., some malware may not trigger the harmful behavior if they detect to be run in emulators).

Regarding the compatibility issue, for this research, the same OS versions were used both in the emulator and the real device (i.e., Android 8.0). As the initial dataset used is the same for both platforms, using the same OS versions (i.e., API level) on both devices ensures that the installation failure on a single device is due to a distinct reason (e.g., not valid certificates or compatible libraries) and grants methodological coherence. However, as stated before, the selection of an OS version always implies that some incompatibilities may have happened (i.e., on both devices) and that threatens the external validity of the results. In any case, the non-execution of an application in the emulator might be an indicator of incompatibility due to distinct reasons than the OS if the application was successfully installed in the real device. Based on this fact, the samples that did not run on the emulator were discarded (i.e., just for the emulator dataset), a common practice used in other approaches dealing with behavioral data issues (Cai and Ryder, 2017; Cai et al., 2020).

When considering the behavior of malware in emulators, it should be noted that specific malware families may include anti-sandbox capabilities so that they may get suspicious about the platform and hide their harmful actions. In our previous experiments (Guerra-Manzanares et al., 2019; Guerra-Manzanares et al., 2019), although we induced learning models by using the behavioral data of all malware types (i.e., regardless of showing such hiding activities or not) obtained from emulators, the results were similar to the data gathered from real devices. In this research, we did not elaborate on the impact of such activities on the collected dataset as such effort is beyond our objective and requires special attention in a separate study. Nevertheless, researchers can use our datasets to identify and compare the behavioral deviations of the same malware on real devices and emulators and provide reasoning about possible hiding capabilities.

7. Conclusions and Future Work

The *changing* nature of Android malware has been neglected by Android malware research and the available datasets, which provide a *rigid* and limited snapshot of Android malware in a restricted time-frame. In general, the *time* variable has never had the deserved attention, disregarding the *concept drift*. Besides, the source of dynamic data and their particularities have been overlooked. However, in order to build more effective, robust, and time-lasting detection systems, the *time* and data *platform source* are critical factors that must be addressed.

In this research, different sources of benign and malware data were merged, enabling to generate a dataset encompassing a larger time-frame, and 489 static and dynamic features were collected. To attend to the particularities of distinct dynamic data sources (i.e., system calls), an emulator and a real device were used, generating two equally-featured datasets. As a result, the main outcome of this research is a novel, labeled, and hybrid-featured Android dataset that provides timestamps for each data sample, covering all years of Android history, from 2008-2020, and considering the distinct dynamic data sources. The emulator dataset is composed of 28,745 malware from 209 malware families and 35,246 benign samples. The real device dataset contains 41,382 malware, belonging to 240 malware families, and 36,755 benign samples. To the best of our knowledge, this is the first research where the applications' *timestamps* and the distinct *platform sources* of dynamic features for Android malware detection are considered. Made publicly available as *KronoDroid*,[2] this dataset is the largest hybrid-featured Android dataset and the only one providing timestamped data, considering dynamic sources' particularities and containing samples of more than 209 malware families.

The detection and characterization of *concept drift*, further investigation of the dynamic differences between emulators and real devices, and evolution of malware families over time constitute part of the authors' plans to be investigated with the assistance of *KronoDroid*.

CRediT author statement

Alejandro Guerra-Manzanares: Conceptualization, Methodology, Software, Investigation, Formal Analysis, Writing-Original Draft, Writing - Review & Editing, Visualization; **Hayretdin Bahsi**: Conceptualization, Methodology, Validation, Writing-Original Draft, Writing - Review & Editing and Supervision; **Sven Nomm**: Conceptualization, Methodology, Writing - Review & Editing and Supervision

[2] https://github.com/aleguma/kronodroid

Declaration of Competing Interest

None.

Appendix A. Real Device Distributions

Fig. A13, A14, A15, A16

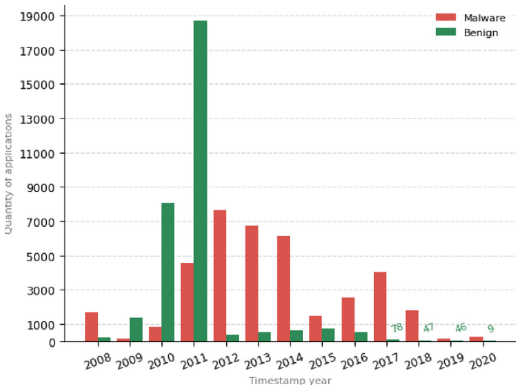


Fig. A.13 – Real device *Earliest Mod valid* year distribution.

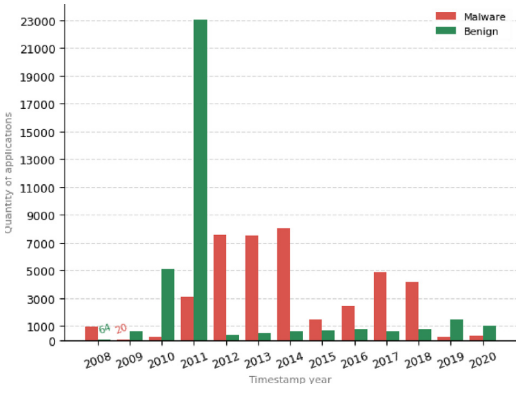


Fig. A.14 – Real device *Last Mod valid* year distribution.

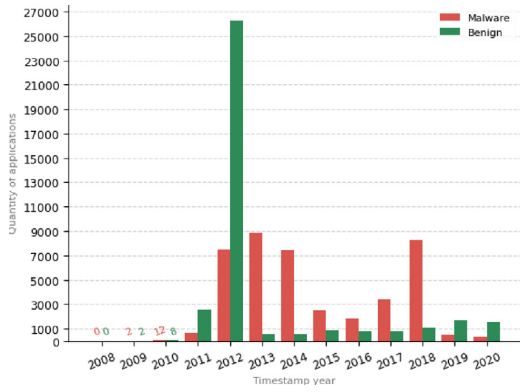


Fig. A.15 – Real device *First Seen VT valid* year distribution.

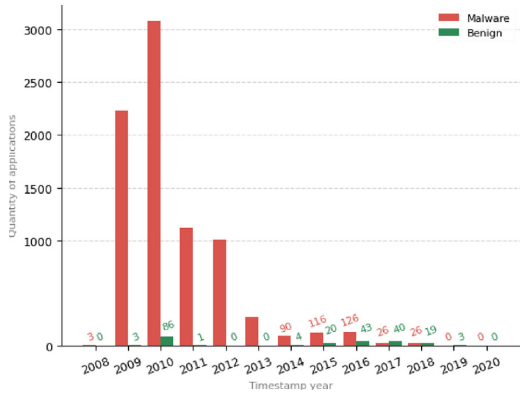


Fig. A.16 – Real device *First Seen ITW valid* year distribution.

REFERENCES

Allix K, Bissyandé TF, Klein J, Traon YLe. Androzoo: Collecting millions of android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16. ACM; 2016. p. 468–71. http://doi.acm.org/10.1145/2901739.2903508. doi:10.1145/2901739.2903508.

Alzaylaee MK, Yerima SY, Sezer S. Dl-droid: Deep learning based android malware detection using real devices. Computers Security 2020;89.

Amos B, Turner H, White J. Applying machine learning classifiers to dynamic android malware detection at scale. In: 2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC); 2013. p. 1666–71. doi:10.1109/IWCMC.2013.6583806.

Android, Android abis, https://developer.android.com/ndk/guides/abis/, 2020.

Android, Application fundamentals, https://developer.android.com/guide/components/fundamentals, 2021.

Android, Reduce your app size, https://developer.android.com/topic/performance/reduce-apk-size, 2021.

Android, Introduction to activities, https://developer.android.com/guide/components/activities/intro-activities, 2021.
Android, Services overview, https://developer.android.com/guide/components/services, 2021.
Android, Broadcasts overview, https://developer.android.com/guide/components/broadcasts, 2021.
Android, Content providers, https://developer.android.com/guide/topics/providers/content-providers, 2021.
Android, How it works, https://developer.android.com/guide/topics/manifest/manifest-intro/, 2021.
Android, Ui/application exerciser monkey, https://developer.android.com/studio/test/monkey, 2021.
Android, <permission>, https://developer.android.com/guide/topics/manifest/permission-element/, 2021.
Android, Aapt2, https://developer.android.com/studio/command-line/aapt2, 2021.
Android, Set the application id, https://developer.android.com/studio/build/application-id, 2021.
Android, Manifest.permission, https://developer.android.com/reference/android/Manifest.permission/, 2021.
APKMirror, Faq - security, https://www.apkmirror.com/faq/#Security_What_measures_do_you_take_to_make_sure_all_uploadedAPKs_are_real_and_created_by_the_respective_developers, 2021.
APKMirror, Apkmirror, https://www.apkmirror.com/, 2020.
ArgusLab, Amd dataset - argus cyber security lab, http://amd.arguslab.org/, 2020.
Arora A, Garg S, Peddoju SK. Malware detection using network traffic analysis in android based mobile devices. In: 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies; 2014. p. 66–71. doi:10.1109/NGMAST.2014.57.
Arp D, Spreitzenbarth M, Hubner M, Gascon H, Rieck K, Siemens C. Drebin: Effective and explainable detection of android malware in your pocket. Ndss 2014;14:23–6.
Arp D, Quiring E, Pendlebury F, Warnecke A, Pierazzi F, Wressnegger C, Cavallaro L, Rieck K, Dos and don'ts of machine learning in computer security, arXiv preprint arXiv:2010.09470 (2020).
Barbero F, Pendlebury F, Pierazzi F, Cavallaro L, Transcending transcend: Revisiting malware classification with conformal evaluation, arXiv preprint arXiv:2010.03856 (2020).
Bl˘asing T, Batyuk L, Schmidt A-D, Camtepe SA, Albayrak S. An android application sandbox system for suspicious software detection. In: 2010 5th International Conference on Malicious and Unwanted Software. IEEE; 2010. p. 55–62.
Bovet DP, Cesati M, Understanding the Linux Kernel: from I/O ports to process management, " O'Reilly Media, Inc.", 2005.
Braunschweig TU, The drebin dataset, https://www.sec.cs.tu-bs.de/~danarp/drebin/index.html, 2020.
Broersma M, Android hit by 'incredibly sophisticated' malware, https://www.silicon.co.uk/workspace/android-sophiticated-malware-344222, 2020.
Burguera I, Zurutuza U, Nadjm-Tehrani S. Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices; 2011. p. 15–26.
Cai H, Jenkins J. Towards sustainable android malware detection. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18, Associa- tion for Computing Machinery; 2018. p. 350–1. doi:10.1145/3183440.3195004.
Cai H, Ryder BG. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). Artifacts for dynamic analysis of android apps; 2017. 659–659. doi:10.1109/ICSME.2017.36.
Cai H, Ryder BG. Understanding android application programming and security: A dynamic study. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME; 2017. p. 364–75. doi:10.1109/ICSME.2017.31.
Cai H, Ryder BG. A longitudinal study of application structure and behaviors in android. IEEE Transactions on Software Engineering 2020 1–1.
Cai Z, Yap RH. Inferring the detection logic and evaluating the effectiveness of android anti-virus apps. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy; 2016. p. 172–82.
Cai H, Meng N, Ryder B, Yao D. Droidcat: Effec- tive android malware detection and categorization via app-level profiling. IEEE Transactions on Information Forensics and Security 2019;14:1455–70.
Cai H, Fu X, Hamou-Lhadj A. A study of run-time behavioral evolution of benign versus malicious apps in android. Information and Software Technology 2020;122.
Cai M, Jiang Y, Gao C, Li H, Yuan W. Learning features from enhanced function call graphs for android malware detection. Neurocomputing 2021;423:301–7.
Cai H. Assessing and improving malware detection sustainability through app evolution studies. ACM Transactions on Software Engineering and Methodology (TOSEM) 2020;29:1–28.
Cai H. In: Embracing mobile app evolution via continuous ecosystem mining and characterization, MOBILESoft '20. New York, NY, USA: Association for Computing Machinery; 2020. p. 31–5. doi:10.1145/3387905.3388612.
Cai H, Tracedroid: Eight-year behavioral profiles of android apps, https://zenodo.org/record/3665877#.YNmvvegzYuU, 2020.
Chebyshev V, Mobile malware evolution 2019, https://securelist.com/mobile-malware-evolution-2019/96280, 2020.
Cimpanu C, Gustuff android banking trojan targets 125+ banking, im, and cryptocurrency apps, https://www.zdnet.com/article/gustuff-android-banking-trojan-targets-100-banking-im-and-cryptocurrency-apps, 2019.
Cohen R, Walkowski D, Banking trojans: A reference guide to the malware family tree, https://www.f5.com/labs/articles/education/banking-trojans-a-reference-guide-to-the-malware-family-tree, 2019.
Cortes C, Jackel LD, Chiang W-P. Limits on learning machine accuracy imposed by data quality. Advances in Neural Information Processing Systems 1994;7:239–46.
Desnos A, Gueguen G, Bachmann S, Androguard, https://androguard.readthedocs.io/en/latest, 2018.
Dini G, Martinelli F, Saracino A, Sgandurra D. Madam: a multi-level anomaly detector for android malware. In: International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security. Springer; 2012. p. 240–53.
Dunham K, Hartman S, Morales JA, Quintans M, Strazzere T. Android Malware and Analysis. 1st ed. USA: CRC Press; 2015.
El Fiky AH. Deep-droid: Deep learning for android malware detection. International Journal of Innovative Technology and Exploring Engineering 2020;9.
Enck W, Ongtang M, McDaniel P. On lightweight mobile phone application certification. In: Proceedings of the 16th ACM conference on Computer and communications security; 2009. p. 235–45.
Enck W, Gilbert P, Han S, Tendulkar V, Chun B-G, Cox LP, Jung J, McDaniel P, Sheth AN. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS) 2014;32:1–29.

F-droid, F-droid - free and open source android app repository, https://f-droid.org/, 2020.
F-Secure, Riskware:android/smsreg.variant!online, https://www.f-secure.com/sw-desc/riskware_android_smsreg_online.shtml, 2021.
Faruki P, Ganmoor V, Laxmi V, Gaur MS, Bharmal A. Androsimilar: robust statistical feature signature for android malware detection. In: Proceedings of the 6th International Conference on Security of Information and Networks; 2013. p. 152–9.
Fedler R, Schu¨tte J, Kulicke M. On the effectiveness of malware protection on android. Fraunhofer AISEC 2013;45.
Feizollah A, Anuar NB, Salleh R, A WA. Wahab, A review on feature selection in mobile malware detection. Digital investigation 2015;13:22–37.
Feizollah A, Anuar NB, Salleh R, Suarez-Tangil G, Furnell S. Androdialysis: Analysis of android intent effectiveness in malware detection. computers security 2017;65:121–34.
Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified. In: Proceedings of the 18th ACM conference on Computer and communications security; 2011. p. 627–638.
Fu X, Cai H. On the deterioration of learning-based malware detectors for android. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion); 2019. p. 272–3. doi:10.1109/ICSE- Companion.2019.00110.
Google, Google play protect, https://developers.google.com/android/play-protect, 2021.
AppBrain, Number of android apps on google play, https://www.appbrain.com/stats/number-of-android-apps, 2021.
Grace M, Zhou Y, Zhang Q, Zou S, Jiang X. Riskranker: scalable and accurate zero-day android malware detection. In: Proceedings of the 10th international conference on Mobile systems, applications, and services; 2012. p. 281–94.
Grace M, Zhou Y, Zhang Q, Zou S, Jiang X. Riskranker: scalable and accurate zero-day android malware detection. In: Proceedings of the 10th international conference on Mobile systems, applications, and services; 2012. p. 281–94.
Guerra-Manzanares A, Nomm S, Bahsi H. In-depth feature selection and ranking for automated detection of mobile malware. ICISSP 2019:274–83.
Guerra-Manzanares A, Bahsi H, N˜omm S. Differences in android behavior between real device and emulator: A malware detection perspective. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS). IEEE; 2019. p. 399–404.
Guerra-Manzanares A, Nõmm S, Bahsi H. Time-frame analysis of system calls behavior in machine learning-based mobile malware detection. In: 2019 International Conference on Cyber Security for Emerging Technologies (CSET). IEEE; 2019. p. 1–8.
Guerra-Manzanares A, Bahsi H, Nõmm S. Differences in android behavior between real device and emulator: A malware detection perspective. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS); 2019. p. 399–404. doi:10.1109/IOTSMS48152.2019.8939268.
Guerra-Manzanares A, Kronodroid dataset, https://github.com/aleguma/kronodroid, 2021.
Hahn K, Malware naming hell part 1: Taming the mess of av detection names, https://www.gdatasoftware.com/blog/2019/08/35146-taming-the-mess-of-av-detection-names, 2019.
Hahn K, Ransomware identification for the judicious analyst, https://www.gdatasoftware.com/blog/2019/06/31666-ransomware-identification-for-the-judicious-analyst, 2019.
Harvey P, Exiftool, https://exiftool.org/, 2021.
Hou S, Saas A, Chen L, Ye Y. Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In: 2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW). IEEE; 2016. p. 104–11.
Hou S, Saas A, Chen L, Ye Y, Bourlai T. Deep neural networks for automatic android malware detection. In: Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017; 2017. p. 803–10.
Hu D, Ma Z, Zhang X, Li P, Ye D, Ling B. The concept drift problem in android malware detection and its solution. Security and Communication Networks 2017;2017.
Hu D, Ma Z, Zhang X, Li P, Ye D, Ling B, The concept drift problem in android malware detection and its solution, Security and Communication Networks 2017 (2017).
Idrees F, Rajarajan M. Investigating the android intents and permissions for malware detection. In: 2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob); 2014. p. 354–8. doi:10.1109/WiMOB.2014.6962194.
Iqbal M, App download and usage statistics (2020), https://www.businessofapps.com/data/app-statistics, 2020.
Irolla P, Dey A. The duplication issue within the drebin dataset. Journal of Computer Virology and Hacking Techniques 2018;14:245–9.
Irolla P, Dey A. The duplication issue within the drebin dataset. Journal of Computer Virology and Hacking Techniques 2018;14:245–9.
Jordaney R, Sharad K, Dash SK, Wang Z, Papini D, Nouretdinov I, Cavallaro L. Transcend: Detecting concept drift in malware classification models. In: 26th USENIX Security Symposium (USENIX Security 17; 2017. p. 625–42.
Jordaney R, Sharad K, Dash SK, Wang Z, Papini D, Nouretdinov I, Cavallaro L. Transcend: Detecting concept drift in malware classification models. In: 26th USENIX Security Symposium (USENIX Security 17; 2017. p. 625–42.
Kabakus AT, Dogru IA. An in-depth analysis of android malware using hybrid techniques. Digital Investigation 2018;24:25–33.
Kadir AFA, Stakhanova N, Ghorbani AA. Android botnets: What urls are telling us. In: International Conference on Network and System Security. Springer; 2015. p. 78–91.
Kaspersky, Rules for naming, https://encyclopedia.kaspersky.com/knowledge/rules-for-naming, 2021.
Kiss N, Lalande J-F, Leslous M, Viet Triem Tong V. In: Learning from Authoritative Security Experiment Results. Kharon dataset: Android malware under a microscope. San Jose, United States: The USENIX Association; 2016 https://hal-univ-orleans.archives-ouvertes.fr/hal-01300752.
Kiss N, Lalande J-F, Leslous M, Viet Triem Tong V, Kharon malware dataset, http://kharon.gforge.inria.fr/dataset, 2021.
Lakshmanan R, Joker malware apps once again bypass google's security to spread via play store, https://thehackernews.com/2020/07/joker-android-mobile-virus.html, 2020.
Lashkari AH, Kadir AFA, Gonzalez H, Mbah KF, Ghorbani AA. Towards a network-based framework for android malware detection and characterization. In: 2017 15th Annual Conference on Privacy, Security and Trust (PST). IEEE; 2017. p. 233–23309.
Lashkari AH, AKadir AF, Gonzalez H, Mbah KF, Ghorbani AA. Towards a network-based framework for android malware detection and characterization. In: 2017 15th Annual Conference on Privacy, Security and Trust (PST); 2017. p. 233–23309. doi:10.1109/PST.2017.00035.
Lashkari AH, Kadir AFA, Taheri L, Ghor- bani AA. Toward developing a systematic approach to generate benchmark android malware datasets and classification. In: 2018 International Carnahan Conference on Security Technology (ICCST). IEEE; 2020. p. 1–7.

Lei T, Qin Z, Wang Z, Li Q, Ye D. Evedroid: Event-aware android malware detection against model degrading for iot devices. IEEE Internet of Things Journal 2019;6:6668–80.
Levin D, Strace - linux syscall tracer, https://strace.io, 2021.
Li D, Wang Z, Xue Y. Fine-grained android malware detection based on deep learning. In: 2018 IEEE Conference on Communications and Network Security (CNS); 2018. p. 1–2. doi:10.1109/CNS.2018.8433204.
Li H, Zhou S, Yuan W, Li J, Leung H. Adversarial-example attacks toward android malware detection system. IEEE Systems Journal 2020;14:653–6.
Li W, Fu X, Cai H, Androct: Ten years of app call traces in android, https://zenodo.org/record/5010831#.YNnJA-gzYuU, 2021.
Liang S, Du X. Permission-combination-based scheme for android mobile malware detection. In: 2014 IEEE International Conference on Communications (ICC); 2014. p. 2301–6. doi:10.1109/ICC.2014.6883666.
Lindorfer M, Neugschwandtner M, Platzer C. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis, 2; 2015. p. 422–33.
Lipovsky´R, S˘tefanko L, Brani˘sa G, The rise of android ransomware, https://www.welivesecurity.com/wp-content/uploads/2016/02/Rise_of_Android_Ransomware.pdf, 2016.
du Luxembourg U, Androzoo, https://androzoo.uni.lu, 2021.
du Luxembourg U, Androzoo - lists of apks, https://androzoo.uni.lu/lists, 2021.
Mahdavifar S, Kadir AFA, Fatemi R, Alhadidi D, Ghorbani AA. Dynamic android malware category classification using semi-supervised deep learning. In: 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Comput- ing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech). IEEE; 2020. p. 515–22.
Mariconti E, Onwuzurike L, Andriotis P, De Cristofaro E, Ross G, Stringhini G, Mamadroid: Detecting android malware by building markov chains of behavioral models, arXiv preprint arXiv:1612.04433 (2016).
Mcdonald J, Herron N, Glisson W, Benton R. Machine learning-based android malware detection using manifest permissions. In: Proceedings of the 54th Hawaii International Conference on System Sciences; 2021. p. 6976.
McGowan E, Another 21 malware apps found on google play, https://blog.avast.com/new-malware-apps-on-google-play-avast, 2020.
Microsoft, Sophisticated new android malware marks the latest evolution of mobile ransomware, https://www.microsoft.com/security/blog/2020/10/08/sophisticated-new-android-malware-marks-the-latest-evolution-of-mobile-ransomware, 2020.
Microsoft, Malware names, https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/malware-naming, 2021.
Narayanan A, Yang L, Chen L, Jinliang L. Adaptive and scalable android malware detection through online learning. In: 2016 International Joint Conference on Neural Networks (IJCNN); 2016. p. 2484–91. doi:10.1109/IJCNN.2016.7727508.
Oberheide J, Miller C, Dissecting the android bouncer, https://jon.oberheide.org/files/summercon12-bouncer.pdf, 2012.
Onwuzurike L, Mariconti E, Andriotis P, Cristofaro ED, Ross G, Stringhini G. Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). ACM Transactions on Privacy and Security (TOPS) 2019;22:1–34.
Parkour M, Contagio minidump, http://contagiominidump.blogspot.com/, 2019.
Peiravian N, Zhu X. Machine learning for android malware detection using permission and api calls. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence; 2013. p. 300–5. doi:10.1109/ICTAI.2013.53.
Pendlebury F, Pierazzi F, Jordaney R, Kinder J, Cavallaro L. TESSERACT: Eliminating experimental bias in malware classification across space and time. In: 28th USENIX Security Symposium (USENIX Security 19; 2019. p. 729–46.
Peng H, Gates C, Sarma B, Li N, Qi Y, Potharaju R, Nita-Rotaru C, Molloy I. Using probabilistic generative models for ranking risks of android apps. In: Proceedings of the 2012 ACM conference on Computer and communications security; 2012. p. 241–52.
Petsas T, Voyatzis G, Athanasopoulos E, Polychronakis M, Ioannidis S. Rage against the virtual machine: hindering dynamic analysis of android malware. In: Proceedings of the seventh european workshop on system security; 2014. p. 1–6.
Rahali A, Lashkari AH, Kaur G, Taheri L, Fran- cois G, Massicotte F. In: 10th International Conference on Communication and Network Security. Didroid: Android malware classification and characterization using deep image learning; 2020.
Samsung, About knox, https://www.samsungknox.com/en/about-knox, 2021.
Schmidt A-D, Detection of smartphone malware (2011).
Sessions V, Valtorta M. The effects of data quality on machine learning algorithms. ICIQ 2006;6:485–98.
Shabtai A, Kanonov U, Elovici Y, Glezer C, Weiss Y. andromaly": a behavioral malware detection framework for android devices. Journal of Intelligent Information Systems 2012;38:161–90.
Sikorski M, Honig A. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. 1st ed. USA: No Starch Press; 2012.
Statista, Development of new android malware worldwide from june 2016 to march 2020, https://www.statista.com/statistics/680705/global-android-malware-volume, 2021b.
Statista, Mobile operating systems' market share worldwide from january 2012 to october 2020, https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009, 2021a.
Stringhini G, Mamadroid source code, https://bitbucket.org/gianluca_students/mamadroid_code, 2018.
Taheri L, Kadir AFA, Lashkari AH. Extensible android malware detection and family classification using network-flows and api-calls. In: 2019 International Carnahan Conference on Security Technology (ICCST). IEEE; 2019. p. 1–8.
Talha KA, Alper DI, Aydin C. Apk auditor: Permission-based android malware detection system. Digital Investigation 2015;13:1–14.
Tam K, Khan SJ, Fattori A, Cavallaro L. Copperdroid: automatic reconstruction of android malware behaviors. Ndss 2015.
U. of New Brunswick, Android botnet dataset, https://www.unb.ca/cic/datasets/android-botnet.html, 2020.
U. of New Brunswick, Android adware and general malware dataset (cic-aagm2017), https://www.unb.ca/cic/datasets/android-adware.html, 2020.
U. of New Brunswick, Android malware dataset (cic-andmal2017), https://www.unb.ca/cic/datasets/andmal2017.html, 2020.
U. of New Brunswick, Investigation of the android malware (cic-invesandmal2019), https://www.unb.ca/cic/datasets/invesandmal2019.html, 2020.
U. of New Brunswick, Cccs-cic-andmal-2020, https://www.unb.ca/cic/datasets/andmal2020.html, 2020.
U. of New Brunswick, Cicmaldroid 2020, https://www.unb.ca/cic/datasets/maldroid-2020.html, 2020.
VirusShare, Virusshare, https://virusshare.com/, 2020.

VirusTotal, Virustotal academic malware samples, http://www.virustotal.com, 2020.
VirusTotal, How it works, https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works/, 2020.
Wang W, Zhao M, Wang J. Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. Journal of Ambient Intelligence and Humanized Computing 2019;10:3035–43.
Wei F, Li Y, Roy S, Ou X, Zhou W. Deep ground truth analysis of current android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer; 2017. p. 252–76.
Wen Li XF, Cai H, Androct: Ten years of app call traces in android, in: The 18th International Conference on Mining Software Repositories (MSR 2021), Data Showcase Track, 2021.
Withwam R, Android antivirus apps are useless here's what to do instead, https://www.extremetech.com/computing/104827-android-antivirus-apps-are-useless-heres-what-to-do-instead, 2020.
Wu Q, Li M, Zhu X, Liu B. Mviidroid: A multiple view information integration approach for android malware detection and family identification. IEEE MultiMedia 2020;27:48–57.
Xu K, Li Y, Deng R, Chen K, Xu J. Droidevolver: Self-evolving android malware detection system. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE; 2019. p. 47–62.
Yang Y, Du X, Yang Z, Liu X. Android malware detection based on structural features of the function call graph. Electronics 2021:10.
Yerima SY, Sezer S, Muttik I. High accuracy android malware detection using ensemble learning. IET Information Security 2015;9:313–20.
Yuan Z, Lu Y, Wang Z, Xue Y. Droid-sec: deep learning in android malware detection. In: Proceedings of the 2014 ACM conference on SIGCOMM; 2014. p. 371–2.
Zhang X, Zhang Y, Zhong M, Ding D, Cao Y, Zhang Y, Zhang M, Yang M. Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security; 2020. p. 757–70.
Zheng C, Xu Z, New android malware family evades antivirus detection by using popular ad libraries, https://unit42.paloaltonetworks.com/new-android-malware-family-evades-antivirus-detection-by-using-popular-ad-libraries, 2015.
Zhou Y, Jiang X. Dissecting android malware: Characterization and evolution. 2012 IEEE Symposium on Security and Privacy 2012:95–109. doi:10.1109/SP.2012.16.
Zhou Y, Jiang X, Malgenome project, http://www.malgenomeproject.org/, 2015.
Zhu Dali, Jin Hao, Yang Ying, Wu D, Chen Weiyi. Deepflow: Deep learning-based malware detection by mining android application for abnormal usage of sensitive data. In: 2017 IEEE Symposium on Computers and Communications (ISCC); 2017. p. 438–43. doi:10.1109/ISCC.2017.8024568.

Alejandro Guerra Manzanares is a Ph.D. candidate at the Center for Digital Forensics and Cyber Security, Department of Software Science, Tallinn University of Technology, Estonia. He received a B.A. degree in Criminology from the Autonomous University of Barcelona, Spain, in 2013, and a B.S. degree in ICT Engineering from the Polytechnic University of Catalonia, Spain, in 2017. In 2018, he received a M.Sc. in Cybersecurity from Tallinn University of Technology, Estonia. His research interests are in the application of machine learning and deep learning techniques to digital forensics and cyber security related issues, such as mobile malware and IoT botnet detection.

Hayretdin Bahsi is a research professor at the Center for Digital Forensics and Cyber Security at Tallinn University of Technology, Estonia. He has two decades of professional and academic experience in cybersecurity. He received his PhD from Sabancı University (Turkey) in 2010. He was involved in many R&D and consultancy projects about cybersecurity as a researcher, consultant, trainer, project manager, and program coordinator at the National Cyber Security Research Institute of Turkey between 2000 and 2014. His research interests include machine learning and its application to cyber security and digital forensic problems.

Sven Nõmm holds the position of senior researcher in the Institute of Software Science at Tallinn University of Technology. In 1995 he graduated from St. Petersburg State University with a diploma in Applied Mathematics (M.Sc. equivalent). He received a PhD degree from Tallinn University of Technology (Estonia) and Ecole Centrale de Nantes et Université de Nantes (France) in 2004. His research interests are spanned around the application of machine learning and artificial intelligence techniques in the areas of medical diagnostics and cybersecurity.

Appendix 5

Publication V
A. Guerra-Manzanares, M. Luckner, and H. Bahsi. Android malware concept drift using system calls: Detection, characterization and challenges. *Expert Systems with Applications*, 206:117200, 2022

Expert Systems With Applications 206 (2022) 117200

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Android malware concept drift using system calls: Detection, characterization and challenges☆

Alejandro Guerra-Manzanares [a,*], Marcin Luckner [b], Hayretdin Bahsi [a]

[a] Department of Software Science, Tallinn University of Technology, Estonia
[b] Faculty of Mathematics and Information Science, Warsaw University of Technology, Poland

ARTICLE INFO

Keywords:
Concept drift
Android malware
System calls
Mobile malware
Malware characterization
Malware detection
Malware evolution
Malware behavior

ABSTRACT

The majority of Android malware detection solutions have focused on the achievement of high performance in old and short snapshots of historical data, which makes them prone to lack the generalization and adaptation capabilities needed to discriminate effectively new malware trends in an extended time span. These approaches analyze the phenomenon from a stationary point of view, neglecting malware evolution and its degenerative impact on detection models as new data emerge, the so-called *concept drift*. This research proposes a novel method to detect and effectively address concept drift in Android malware detection and demonstrates the results in a seven-year-long data set. The proposed solution manages to keep high-performance metrics over a long period of time and minimizes model retraining efforts by using data sets belonging to short periods. Different timestamps are evaluated in the experimental setup and their impact on the detection performance is compared. Additionally, the characterization of concept drift in Android malware is performed by leveraging the inner workings of the proposed solution. In this regard, the discriminatory properties of the important features are analyzed at various time horizons.

1. Introduction

Android operating system (OS) leads the mobile OS market since 2012. At present, over 71% of smartphones are powered by this open-source, highly customizable, and versatile OS (Statista, 2021c). Its ubiquity combined with the open-source nature of the OS, the high prevalence of devices running outdated OS versions, the poor end-user security awareness (e.g., *sideloading* and running over-privileged apps), and the wealth of data stored in these devices make Android users an attractive target for cyber attackers (Rafter, 2021). Despite the security enhancements introduced in the OS regular upgrades, Android is still the most targeted mobile operating system by malware, accounting for over 98% of the mobile cyber attacks (Kaspersky, 2020). These attacks are carried out using a wide variety of attack vectors over the large attack surface exposed by mobile devices (Townsend, 2020). In 2020, an average of 482,579 new Android malware samples were discovered per month (Statista, 2021a). Mostly *trojans* and *adware*, the most predominant malware types nowadays (Statista, 2021b; Chebyshev, 2021). However, the threat landscape is not static but subject to continuous change. For instance, ransomware Trojans were the most predominant type of Trojans in 2017, whereas, in 2020, banking Trojans were significantly more prevalent (Unuchek, 2018; Chebyshev, 2021). New malware trends have emerged over time and more sophisticated malware samples have been discovered, evidencing the non-stationary attribute of the threat, featured by constant evolution and innovation (Microsoft, 2020). As a result, the figures regarding detected mobile malware may only reflect a small portion of the total malware *in the wild*, with new and more sophisticated malware variants remaining undetected (Broersma, 2020).

Notwithstanding the dynamic nature of the phenomenon, the specialized research has overlooked the evolution and changes in Android malware over time. In this regard, despite the vast body of literature available on the optimization of detection methods, the change in malware features over time and its degenerative impact on the machine learning-based detection models, the so-called *concept drift*, has not been explored thoroughly. Most machine learning-based models for malware detection are based on the assumption of *consistent* data, thus requiring the properties of the testing data distribution to approximately

☆ The code (and data) in this article has been certified as Reproducible by the CodeOcean: https://codeocean.com/capsule/6162888/tree/v1. More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail addresses: alejandro.guerra@taltech.ee (A. Guerra-Manzanares), mluckner@mini.pw.edu.pl (M. Luckner), hayretdin.bahsi@taltech.ee (H. Bahsi).

https://doi.org/10.1016/j.eswa.2022.117200
Received 12 September 2021; Received in revised form 7 April 2022; Accepted 7 April 2022
Available online 21 April 2022

match the characteristics of the training data distribution. However, due to the constant battle between attackers and defenders, malware evolve to exploit new vulnerabilities and improve its hiding capabilities in response to enhanced defenses, thus generating new malware variants that may use a distinct attack vector and behave differently but pursue the same ends. As a result, the incoming data distribution may diverge significantly from the model's original training distribution, thus, generating concept drift and, consequently, harming the model's performance over time. Despite that, the myriad of machine learning-based solutions proposed for Android malware detection are generally built, validated, and tested using relatively small data sets, collected in short time-frames, generally far from the present time. For instance, the most used Android data sets in research studies were gathered between 2010 and 2012 (i.e., MalGenome (Zhou & Jiang, 2012) and Drebin (Arp et al., 2014)), a decade ago, when malware capabilities and characteristics were significantly distinct to the present ones (e.g., the first Android ransomware was detected in 2014 and had its major outbreak in 2016, ergo not represented in these data sets). Even so, they are still being used in recent studies as the main malware references and frequently as the only malware source (Zhao et al., 2021; Reddy et al., 2021). Therefore, most of the proposed solutions have been optimized for malware detection at specific snapshots of the Android history, built and tested on *static* and *partial* data sets containing specific malware trends at a particular time, thus lacking the generalization capabilities needed to address the constant evolution of Android malware, its non-stationary character, and its challenges.

The majority of studies on Android malware detection based on machine learning techniques lack *historical coherence*, mainly caused by splitting randomly the available data into the *testing* and *training* sets without respecting the *historical timeline*. More specifically, the testing set should always be composed of *future* or *posterior* data regarding the training set (Allix et al., 2015; Arp et al., 2020). In these solutions, concept drift is neglected and the optimized models provided by such approaches yield *significantly biased* and *historically incoherent* results, a critical issue when it comes to real Android malware detection (Allix et al., 2015; Pendlebury et al., 2019; Arp et al., 2020). In this regard, a critical challenge to deal with concept drift is how to locate the samples within the Android historical timeline in a reliable way. Our research uses and compares two timestamping approaches that may result in good approximations in the search for temporal accuracy. Furthermore, in production setups, the number of samples processed may rarely be evenly split between the classes. For instance, a malware outbreak may cause the processed data to be imbalanced towards the positive class (i.e., malware label), whereas, in the absence of an outbreak, the majority of the new samples should be benign. This fact is usually not considered by the related research, assuming and working with evenly split data sets. Our study addresses the concept drift issue in the presence of imbalanced data, providing a more realistic scenario and reliable results. Thus, the proposed solution can effectively handle the additional challenge of imbalanced data towards any of the classes. Lastly, the small number of studies that proposed solutions considering Android malware concept drift issues did not provide any insights on the changes in the data, that is, the characterization of concept drift. This is a distinctive point of this study, which not only addresses concept drift but also aims to understand the evolution of features over time in the analyzed context. The characterization of concept drift allows understanding the direction of changes, enabling the expansion and enhancement of the knowledge about the threat while providing useful insights to improve the detection systems.

1.1. Novelty and contribution

In this paper, we address significant research gaps in Android malware detection studies, by exploring, addressing, and characterizing the phenomenon of concept drift in Android malware detection using dynamic features (i.e., system calls) on imbalanced data sets. Despite the existence of methods to detect *drifting* data (Yang et al., 2021; Jordaney et al., 2017) and a large body of research regarding malware detection (Liu et al., 2020), just a few studies related to Android malware detection have considered concept drift in their detection solutions (Xu et al., 2019; Onwuzurike et al., 2019; Cai et al., 2018; Hu et al., 2017; Narayanan et al., 2017). All these studies used API calls, a static feature sensitive to code obfuscation and encryption techniques (Kaspersky, 2021). Unlike these previous works, our study uses system calls, runtime data features that are robust to code obfuscation and encryption methods. System calls enable us to capture the real behavior of the app and are the most used dynamic features for Android malware detection (Liu et al., 2020). Besides, no previous study in the field has provided characterization of concept drift nor compared the performance of distinct timestamps when dealing with emerging concept drift, which are unique contributions of this research. Lastly, the usage of the *KronoDroid* data set (Guerra-Manzanares et al., 2021) enables us to overcome the limitations of other data sets and explore concept drift as it provides labeled and timestamped data for the whole Android history (i.e., 2008–2020). In this regard, our analysis spans a seven-year-long continuous time frame, whereas previous works used either discontinued data sets (Hu et al., 2017; Narayanan et al., 2017) or encompassed a shorter time period (Onwuzurike et al., 2019; Xu et al., 2019; Cai et al., 2018). Our workflow is composed of three stages where we analyze and demonstrate the presence of concept drift in Android malware detection, propose a solution to handle it, and characterize its behavior. Furthermore, the performance of the proposed solution is compared with the state-of-the-art solutions, outperforming all of them.

The main novel points of this research are: (1) the usage of system calls as features in an Android concept drift-related study, (2) the proposed solution, which addresses the impact of concept drift on the classification model, enabling the detection system to sustain high detection performance over an extended period of time, even when imbalanced data are present, (3) the characterization of concept drift, which allows the overall understanding of its behavior and direction, and (4) the evaluation of distinct timestamping approaches to effectively deal with concept drift issues.

The paper is structured as follows: Section 2 provides background information and a summary of related research studies in the field. Section 3 explains the methodology followed in this study and introduces the proposed solution to address and characterize concept drift in Android malware detection. Section 4 describes the results of the experimentation using the proposed solution and the main outcomes of this research. Section 5 provides a discussion of the main results and outlines future work. Finally, Section 6 summarizes the study.

2. Background information and related work

2.1. Background information

A *data stream* can be defined as a *countably infinite sequence of elements* that become available over time (Margara & Rabl, 2018). Due to their cumulative, continuous, rapid, and evolving nature, data streams, usually referenced under the umbrella term of *big data*, pose a variety of challenges such as one-pass constraint, concept drift, resource restriction, and massive-valued features (Aggarwal, 2015). Despite not facing all these challenges, Android malware detection shows issues related to data stream processing such as large data volume, continuous release of apps, and evolving data. Consequently, Android malware concept drift may be effectively handled when tackled from a data stream perspective.

This paper performs a novel attempt to demonstrate, handle and characterize Android data concept drift using system calls as model features and from a data stream perspective. For this purpose, state-of-the-art algorithms are leveraged and customized in our study to tackle effectively concept drift issues in Android malware detection. The following paragraphs introduce their basics.

Gözüaçık and Can (2020) proposed an implicit (i.e., unsupervised) learning algorithm called *One-Class Drift Detector* (OCDD), which uses a one-class learner with a sliding window to detect concept drift. As the data analyzed in our work do not show normal distribution characteristics, it was not possible to apply statistical analysis to detect changes in the features over time. Therefore, the *OCDD*'s central idea was leveraged to analyze the impact of concept drift in the observed data. This approach was implemented in our work using the *Isolation Forest* algorithm. Isolation Forest (iForest) is an anomaly detection technique proposed by Liu et al. (2012). The algorithm uses binary decision trees to detect anomalous data based on path length. More precisely, in randomly generated binary trees, where instances are recursively partitioned, these trees produce noticeable shorter paths for anomalies. In the regions occupied by anomalies fewer partitions are observed (i.e., shorter paths in the tree structure). For a specific sample, the received path length is compared to the average path length of unsuccessful searches in the binary search tree to obtain a universal anomaly measure. This measure is used by the *iForest* algorithm to detect anomalies based on the results obtained on several trees.

Zyblewski et al. (2021) proposed a novel framework employing stratified bagging to train base classifiers, integrating data preprocessing, and using dynamic ensemble selection methods for imbalanced data stream classification. The experimental results showed that dynamic ensemble selection coupled with data pre-processing could outperform state-of-art methods for highly imbalanced data streams. In our work, we analyze Android app data in an extended time frame, where the ratio of malware to benign applications varies over time, thus causing imbalanced data issues. The high-level framework proposed by Zyblewski et al. (2021) was the point of departure of our algorithm and its application to the Android data issue. The original algorithm, designed for data streams split into *data chunks*, uses a pool of classifiers trained on past data to classify new data samples in upcoming data chunks. The best combination of classifiers for the new data (i.e., an ensemble of classifiers) are dynamically selected using the previous data chunk. The classifier pool is constantly purged and updated with new classifiers trained on data from each new chunk. Concept drift is addressed by the constant update of the pool of classifiers, while the ensemble selection mechanism enhances the classification results. The original algorithm was modified in our work to address the particularities of Android data and enhance the detection results, as reported in Section 3.2.2 and Section 4.3.

2.2. Related work

A large variety of malware detection approaches have been proposed since the early years of Android OS (Liu et al., 2020). Most of these solutions were optimized and tested on static snapshots of Android malware historical data, using *old* and short-time data sets (Zhou & Jiang, 2012; Arp et al., 2014). As a result, these solutions disregard the evolution and change in data over time and its potentially harmful effect on the detection system's performance.

The phenomenon of *concept drift*, where the statistical characteristics of the incoming data change over time, is visible in long-term Android data (Ramirez-Gallego et al., 2017). Neglecting the changes in malware data patterns over time has a significant detrimental impact on the classifiers' performance, as models built using *old* data tend to make poor and ambiguous decisions when tested on *new* data (Jordaney et al., 2017). Thus, adapting to the rapid evolution of Android malware is critical for an effective detection system (Hu et al., 2017). Consequently, concept drift should be considered in all ML-based detection methods aiming to provide high and reliable performance over time. However, even though an increasing number of studies recognize the importance of addressing concept drift in Android malware detection models (Suarez-Tangil et al., 2017; Hu et al., 2017), only a reduced number of studies have taken its impact into account. These solutions are briefly discussed in the following paragraphs.

Hu et al. (2017) proposed the usage of an ensemble of classifiers to analyze data within a sliding window and dynamic adjustments to address concept drift on static features (i.e., permissions, actions, and selected API calls). The authors reported 96% accuracy in a relatively small, imbalanced, and discontinued in time data set. The time range of the data set and the source of the majority of the samples are not reported, which generates concerns about the results and the actual existence of concept drift in the data, which is assumed but not proved. In our study, the first stage aims to prove the existence of drifting data within the data set. After, the proposed solution is tested on a large and time-extended data set addressing imbalanced data issues.

In *DroidOL* (Narayanan et al., 2016), online algorithms were used to deal with concept drift. The solution was built and tested on a static-featured data set spanning 8 months (i.e., using inter-procedural control-flow graphs as features). The authors reported 84% accuracy on a balanced data set. Even though the usage of an online learning algorithm can have benefits over batch learning algorithms for concept drift handling purposes, it is questionable that the time span of the data set might be too short for the emergence of concept drift, which was assumed but not proved in the study. The usage of online algorithms for concept drift handling was enhanced in *DroidEvolver* (Xu et al., 2019), where a pool of 5 online classifiers was used to build the detection system. Raw API calls were extracted from the source code and used to generate the input vector. After an initialization step, the pool of classifiers was used to label every new instance. Next, based on a *drift* indicator, the detection models and feature sets were updated, if needed. The update of the feature sets (i.e., done incrementally by including all the new API calls) and the update of aging classifiers are intended to provide resilience against concept drift. Besides, the usage of a pool of classifiers aims to avoid the bias of a single classifier and generate more reliable detection results. The usage of a pool of classifiers to improve the detection performance is also leveraged in our proposed solution. However, the distinctive elements of our proposed solution that uses enhanced classifier dynamics (i.e., *dynamic ensemble selection*) instead of online learning algorithms which require constant retraining, the usage of dynamic features instead of static features, and a reduced and stable feature set as opposed to the incremental cost of an ever-growing feature set, provide increased and more stable long-term detection performance, as it is shown in Section 5.

TRANSCEND (Jordaney et al., 2017) framework used statistical metrics to identify when a classification model was consistently misclassifying new data, thus signaling the emergence of concept drift and the aging of the model. In the study, the framework was merely used as a *drift* indicator, not proposing any solution to handle concept drift distinct from data relabeling and model retraining once *drifting* was identified. The limitations of this work were addressed in *TRANSCENDENT* (Barbero et al., 2020), a model agnostic rejection framework composed of conformal evaluators. In the experimental setup, the framework helped to extend the effectiveness of *Drebin* classifier (Arp et al., 2014), doubling its lifespan, keeping an F1 score over 80% for two years.

MaMaDroid (Onwuzurike et al., 2019) used static analysis to extract sequences of API calls from the call flow graph, and abstracted each API call to three distinct higher abstraction levels (i.e., family, package, or class). The sequences of abstracted API calls were used to build the feature vectors, represented using Markov chains. Concept drift was tackled under the assumption that the representation of the sequences of API calls in the higher level of abstraction changes less over time than the raw API calls, thus being more robust and resilient than the approaches that use directly the API calls, such as DroidEvolver (Xu et al., 2019), which need constant retraining, especially after new Android API releases and API changes, to address concept drift.

API-Graph (Zhang et al., 2020), aims to enhance API call-based detection systems by leveraging API semantics and similar API usages among malware. The framework builds an API-level relation graph by extracting entities such as APIs and permissions and establishing their

relations into five *meta*-categories. The authors showed that the usage of *API-Graph* may improve the generalization capabilities and robustness against performance decay of existing solutions such as *MaMaDroid* and *DroidEvolver*.

Event groups semantics were employed in *EveDroid* (Lei et al., 2019). In this study, API call graphs were used in conjunction with event grouping techniques (i.e., clustering) to train neural network models for Android malware detection in IoT devices. The ability of neural networks to extract semantics from the input features was leveraged to provide robustness against performance decay. Even though the study results reported high performance, the two data sets used encompassed short and discontinued time-frames (i.e., 2013–2014 and 2017–2018). More importantly, the models were trained using random selection of samples, thus disregarding the chronological order and mixing the data in both the training and testing sets. Furthermore, legitimate and malware samples did not belong to the same exact time-frames (e.g., legitimate samples were from 2014 and 2018, whereas malware was from 2013, 2014, 2017 and 2018-Q1). As a result, the robustness of the solution against concept drift poses severe doubts.

Even though API calls are considered static features in malware analysis, they can also be collected dynamically (i.e., at run-time). This was the approach taken in *DroidSpan* (Cai, 2020) to overcome the limitations of static features (e.g., code obfuscation and encryption). In this work, particular API calls, related to sensitive data access and operations, invoked at run-time were collected. The data was used to generate an input vector composed of 52 features. Although the solution aimed for long-term stability and reported better results than MaMaDroid, it did not provide any adaptive measures to maintain long-term performance (i.e., the model is never updated), making it prone to concept drift-related performance decay over time. A similar approach was used in Fu & Cai (2019) and Cai et al. (2018).

Pendlebury et al. (2019) emphasize the significant impact of *spatial* and *temporal bias* in the Android malware detection research literature, which has consistently yielded not representative and overly inflated unrealistic performances. More specifically, *temporal bias* is caused by incorrect temporal splits on the training and testing sets (i.e., neglecting concept drift and providing *impossible* temporal configurations), whereas *spatial bias* is caused by unrealistic distributions on the training and testing sets (e.g., class-balanced data sets). The proposed tool, *TESSERACT*, did not provide any novel strategy to tackle concept drift (i.e., incremental retraining, active learning, or classification with rejection), but aimed to assess the robustness of other solutions to performance decay by removing the spatio-temporal bias from the evaluations, thus revealing their true performance. The framework was tested with samples from *AndroZoo* data set (Université du Luxembourg, 2021) which were temporally located using the *dex date* timestamp, suggested by the authors as the most reliable timestamp and comparable to VirusTotal's *first seen* timestamp. The *dex date* timestamp informs about the compilation date of the *apk* (i.e., app's archive file). However, according to Université du Luxembourg (2021), the *dex date* timestamp is no longer a reliable timestamp as the vast majority of the apps released nowadays have a 1980 *dex date*, thus being not usable for temporal location purposes. Our proposed solution uses a distinct and more reliable timestamp, the *last modification*, which is compared in this study to VirusTotal's *first seen* timestamp, taking advantage of the timestamps provided by the *KronoDroid* data set.

As can be observed, the vast majority of proposed solutions focused on static features which suffer from proven limitations to counter-detection techniques, such as code-obfuscation, packing, and encryption (Aghakhani et al., 2020). Although more complex and time-consuming to acquire, dynamic features are more robust against deception techniques, thus preferred for an effective detection solution. System calls, the most used dynamic features for Android malware detection are used in our study (Liu et al., 2020). Besides, none of the studies that dealt with concept drift provided further exploration of the phenomenon (i.e., characterization). In this regard, the characterization of concept drift may provide a better understanding of the phenomenon and assist malware specialists to understand changes in malware over time, detect trends and build more effective detection systems while expanding the knowledge regarding Android malware behavior. An effective solution for Android malware detection should consider the emergence of concept drift and have adaptive skills to change its inner structure according to the detected data changes. In addition, other challenges overlooked by the specialized research should be addressed such as imbalanced data sets and high-dimensionality problems (i.e., ever-growing feature sets). All these issues are considered in this study and addressed by the proposed solution to build a long-lasting, effective, and robust Android malware detection system.

3. Material and methods

3.1. Data set

The data set used in this research is *KronoDroid* (Guerra-Manzanares et al., 2021). This data set is composed of two device-related sub-datasets (i.e., emulator and real device data sets) containing both benign and malware samples. Every sample in the data set (i.e., Android *apps*) is labeled and characterized by 289 dynamic features and 200 static features (i.e., including timestamps). The data set provides data covering the whole historical timeline of Android OS, from 2008 to 2020. In this regard, four timestamp features provide the possible *temporal context* of the apps, which makes this data set the only publicly available Android data source suitable for the investigation of concept drift in Android malware. For this research, just the real device data set, composed of 78,137 samples, was used due to its larger size (i.e., 41,382 malware samples belonging to 240 malware families and 36,755 benign apps).

To analyze the phenomenon of concept drift in Android malware detection from a dynamic perspective, just the dynamic features provided by the *KronoDroid* data set were used along with the class labels and timestamp features. The dynamic features provided by the data set are *kernel* or *system* calls (*syscalls* for short). Syscalls are the most used dynamic features in Android malware detection systems (Liu et al., 2020). The feature set is composed of 288 numeric variables, whose values provide the absolute frequency of each system call invoked by the app during the execution time (i.e., 1 min, no user interaction). Regarding the label, *Kronodroid* provides two labels for each data sample: the *hard label* and the *soft label*. The only difference between them is the labeling approach used, as they are both defined as non-probabilistic binary labels (i.e., $y \in \{0,1\}$). More precisely, the *soft label* relies on the data source to assign the class to an instance without any further verification. The *hard label* is based on a stricter labeling technique as it imputes the class of a sample according to the detection results from *VirusTotal*'s antivirus scanner disregarding the data source, as explained in Guerra-Manzanares et al. (2021). Due to this stricter class imputation, in this study, the *hard label* was selected as it increases the certainty about the class of the samples (i.e., benign or malware), thus increasing the reliability of the concept drift analysis results.

Finally, as the study of concept drift explores the evolution of data over time, the timestamp used to date the samples becomes of critical importance. In this regard, from the four possible timestamps, the *last modification* and *first seen* timestamps were selected as they provide more data coverage, reliability, and accurate location of the apps within the Android historical timeline. The *last modification* timestamp provides the date of the most recent modification of any file inside the app, while the

first seen timestamp reports about the date when the app was submitted for the first time to *VirusTotal*'s antivirus engine.

As a result, 78,137 Android apps described by 288 system calls numeric variables, the label, and two distinct timestamps were used as input data for this research. The following paragraphs explain the workflow followed to effectively detect, handle and characterize concept drift across Android OS history.

3.2. Workflow

The methodology used in this study is composed of three sequential stages related to concept drift: *detection, handling* and *characterization*. At each stage, different techniques are used, which are explained in detail in the following sections. Briefly, the *concept drift detection* stage aims to prove the existence of concept drift in the data using state-of-the-art techniques. Upon confirmation of the presence of concept drift, the following stages are set to address the related issues. In the *concept drift handling* stage, distinct techniques are used to effectively handle concept drift over time. The algorithm used in this stage to reduce the classification error caused by concept drift provides also the information used for *concept drift characterization*, the last stage of the workflow. It is worth emphasizing that the *handling* and *characterization* stages use a distinct methodology than the *detection* stage as the only goal of the latter is to prove the need for concept drift handling techniques in the Android malware detection case.

3.2.1. Concept drift detection

The concept drift detection process consists of three sequential phases, namely *data pre-processing*, *feature selection*, and *drift detection*. The whole process is depicted in Fig. 1 and explained as follows.

The *initial* set of features was pre-processed using a sequential procedure to remove features that might disturb or not provide any significant input data to the machine learning algorithm (i.e., null-valued and redundant features). The outcome was a *refined* set of features obtained after performing the following steps.

1. *Feature variance analysis*: Homogeneous (i.e., zero variance) and zero-valued features (i.e., not invoked by any app) were removed from the initial feature set.
2. *Feature correlation analysis*: *Pearson's correlation coefficient (r)* was calculated pairwise for all variables. Highly correlated features were removed from the feature set.
3. *Feature distribution analysis*: statistical normality tests were performed on all features that remained in the feature set after the previous steps.

The goal of these sequential steps was to select relevant features, remove redundant variables and assess the data distribution to select the best techniques for the posterior steps.

Concept drift detection can be formulated as follows. Each new observation is represented by $c_i = (x_i, y_i)$, where $x_i = (x_i^1, x_i^2, .., x_i^n) \in \mathbf{X}$ is the feature vector and $y_i \in \mathbf{Y}$ is the target label. Given two data chunks (i.e., probes of the same size or collected during similar periods) taken from distributions F and F', respectively, the following applies. For the null hypothesis, H_0, that F and F' are identical, the aim is, as stressed by Lu et al. (2014), to refuse H_0, and identify some local regions of the problem space where H_0 does not hold, and quantify the difference between F and F'. However, as shown by Mutz et al. (2006) and Ruiz-Heras et al. (2017), Android system calls' distributions must be modeled and Gaussian distributions cannot be assumed. For this reason, the analysis of F and F' distributions is hindered.

In addition, the difference between F and F' might be statistically important but not change the decision of the malware detection system. More precisely, concept drift changes the classification decision by changing the conditional probability $p(y|\mathbf{X})$. However, it is possible to change the probability $p(\mathbf{X})$ without changing $p(y|\mathbf{X})$. In such a case, the change in the feature space does not affect the classification outcomes and is called *feature drift* or *virtual drift* (Ramirez-Gallego et al., 2017). The main difference is that under *real concept drift*, the restructuring of the learning model is required, whereas, under virtual drift, the old knowledge is extended with additional data from the same environment (Gözüaçık & Can, 2020; Elwell & Polikar, 2011). Consequently, concept drift is only observed if and only if the input data changes affect the classification results (i.e., the conditional probability $p(y|\mathbf{X})$ is affected).

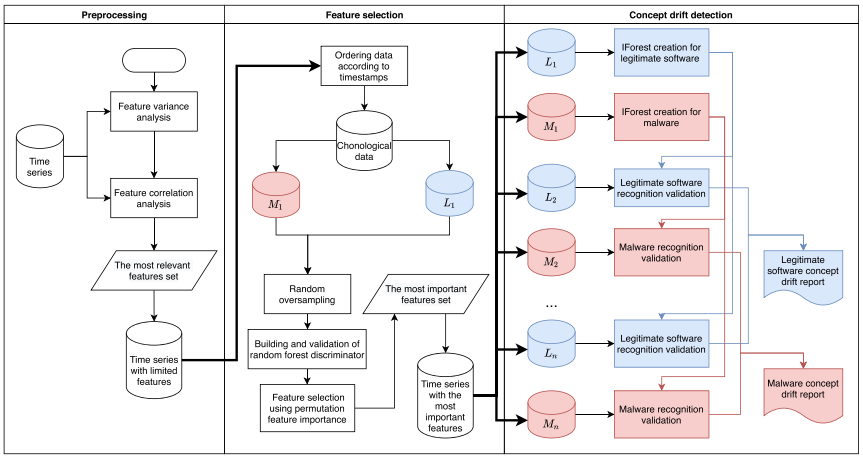


Fig. 1. Schema of the concept drift detection stage.

To assess the existence of concept drift, the data set must be chronologically ordered and divided into consecutive periods. For n periods, two series of subsets are generated. In the first series, the set M_i consists of records describing malware and labeled with the timestamp (i.e., *last modification* or *first seen*) from the i-th period. The set L_i, defined for the second series, is labeled analogously but composed of just benign apps.

Then, the following procedure was performed to define the most discriminatory features for the first period (i.e., feature selection).

The data from the first period $M_1 \cup L_1$ was balanced using a random oversampling method to avoid over-representation of L_1 or M_1 (Seiffert et al., 2010). Next, *Random Forest* (RF) algorithm, a decision tree-based ensemble classification algorithm proposed by Breiman (2001), was induced to discriminate between L_1 and M_1. The rationale behind the selection of RF algorithm is the superior performance shown by this algorithm over other classification algorithms in analogous cases on previous research (Guerra-Manzanares et al., 2019a; Guerra-Manzanares et al., 2019b; Guerra-Manzanares et al., 2019c). The most relevant features of this initial classifier were selected using the *permutation feature importance* technique applied to the training data (see Section 3.2.3 for further details). This enabled the selection of the most relevant features for this baseline classifier. In this regard, only the features with positive mean importance for the model were selected, defining the *important* feature set.

If the initial-period classification model provides high performance (e.g. accuracy $> 95\%$), it is reasonable to assume that the selected features are significant to recognize classes L_1 and M_1. However, an open challenge is whether the same features can be successfully used to recognize classes L_i and M_i for $n \geq i > 1$. To address this issue, One-Class Drift Detection Models (Gözüaçık & Can, 2020) were used to analyze the impact of concept drift in the generated series. Therefore, based on the fact that the selected *important* features could be used successfully for the classification task on the $M_1 \cup L_1$ set, one-class anomaly detectors were induced using L_1 and M_1 separately, to assess data drift. The Isolation Forest (iForest) algorithm was used to induce the anomaly detection models. This approach allows the observer to analyze the concept drift for malware and legitimate software in a more controlled way eliminating the class relations influence.

Next, the iForest detection models induced were tested separately with L_i and M_i sets, where $n \geq i > 1$, described by the *important* feature set to calculate the ratio of samples recognized as part of the modeled class in each period (i.e., negative detection rate). The resulting ratios enable us to assess the emergence of concept drift on the data over time. More specifically, if the initially selected *important* features are not able to describe the modeled class effectively in a given period, the ratio will drop, thus indicating data drift. The observed drops may be qualified as concept drift as the ratio of correctly recognized observations decreases. Therefore, the change affects the classification results.

The workflow performed for concept drift detection is described in Fig. 1 for further reference.

The emergence of concept drift in the data requires the implementation of an adaptive detection solution capable of handling it effectively. This issue is addressed in the next section.

3.2.2. Concept drift handling

The proposed solution to handle concept drift in Android malware is a modification of the algorithm proposed by Zyblewski et al. (2021). Although the original algorithm may provide good performance, the modifications performed address Android concept drift particularities, thus boosting the detection performance. The suggested adaptations are detailed in the following paragraphs.

The first modification yields *a complete pool of classifiers available from the initial stage*. At the initial stage, the original algorithm starts with a single classifier in the pool. For every subsequent data chunk processed, a new classifier is added to the pool until the pool is full. Only then the full pool is available for prediction. For instance, if the size of the pool of classifiers is set to contain 10 elements (i.e., S parameter), then 10 chunks must be processed to have available the full pool of classifiers. Thus, these 10 *initial* data chunks are all tested with an incomplete pool of classifiers. After the pool is completed, in the subsequent data chunks (i.e., from the 11th chunk), each classifier is tested and the worst performer is eliminated (i.e., pool purge). Then, a new classifier trained on the new data chunk is added (i.e., pool update). Therefore, only after the initial S chunks are processed, the pool is guaranteed to be always composed of S elements. Our experimentation has shown that it may be beneficial to generate the full pool from the initialization phase (i.e., first chunk), as the usage of a larger variety of classifiers to process the initial data chunks enhances the performance on the initial stages. As a result, the proposed solution eliminates the delay on the generation of the full pool by splitting the initial chunk into S ordered subchunks and training a classifier on each subset, thus generating the full pool in the initial step. Then, after the first chunk is processed (i.e., initialization stage), the whole pool, containing S elements, is used to process all the subsequent data chunks (i.e., from the 2nd chunk). The classifier pool is updated using the same *purge-update* mechanism as the original solution when a new data chunk is processed.

The second modification performs a *refinement on the predictions using a supportive anomaly detection model*. The original solution produces its predictions according to the classifiers' pool assignment. The proposed modification adds a refinement step to the prediction process by using an anomaly detection model, induced on a subset of data (i.e., benign data), aiming to improve the predictions provided by the dynamic ensemble selection of classifiers. The rationale behind the addition of an anomaly detection model trained on legitimate data is due to the observation of more concept drift resilient features in this subset of data during our experimentation, as shown in Fig. 3 and Fig. 9a. More specifically, concept drift in legitimate data appears to be less significant than in malware data, showing a more robust performance over time and keeping as *important* a broader and more consistent subset of features. Therefore, the addition of an anomaly detector trained on just benign data can help in dubious cases where the prediction probability of the classifier pool may not give enough confidence to the assigned label (e.g., $p(y|x) \approx 0.5$). In such cases, the usage of the additional knowledge from the anomaly detection model might help. However, the benefits of this modification heavily rely on the prediction probabilities output by the classification model and the specific set of rules and thresholds used by the particular implementation.

As an example, in our experimental setup, a *reassignment rule* was applied for borderline predictions as follows: if $0.55 \geq p_{pool}(benign|x) \geq 0.5$, where x refers to a given sample and p_{pool} to a particular prediction probability from the pool of classifiers, then the sample was assigned to the class suggested by the anomaly model. In any other case, the class assigned to x was the one suggested by p_{pool}. This simple rule yielded from 1% to 4% improvement in detection performance in some time periods, especially in the initial chunks. It is worth mentioning that, in our experimental setup, no rule optimization was performed, thus there is room for improvement in this regard by the particular implementations of the proposed solution.

The proposed solution is provided in Algorithm 1 and Algorithm 2. For the sake of comprehension and similarly to the original formulation by Zyblewski et al. (2021), the algorithm is split into *training* and *prediction* phases. However, as these phases are applied sequentially for each data chunk, they should be embedded in the actual implementation of the solution. More specifically, Algorithm 1 provides the *pseudo-code* implementation of the *training* phase while Algorithm 2 defines the steps performed in the *prediction* phase.

Algorithm 1: Training phase of the proposed framework

Input:

Stream - Data stream

S - Fixed size of the classifier pool

$\Pi \leftarrow \varnothing$ - Pool of classifiers (initially empty)

$\lambda \leftarrow -1$ - Sample size of the anomaly detector

Symbols:

DS_k - Data chunk

Ψ_k - Bagging classifier

L_k - Legitimate data portion of the data chunk

Φ_k - Anomaly detector

1 **foreach** k, *DS* ***in*** *Stream* **do**
2 **if** $k == 0$ **then** // first data chunk
3 $IDS \leftarrow splitInitialDataset(DS_k, |\pi|)$ // split data chunk
4 **for** $i \leftarrow 0$ *to* $S - 1$ **do**
5 $\Psi_k \leftarrow trainClassifier(IDS_i)$ // train classifier
6 $\Pi \leftarrow \Psi_k$ // add classifier to the pool
7 **end**
8 $\Phi_k \leftarrow trainAnomalyDetector(L_k, \lambda)$ // train anomaly
9 **else** // rest of the data
10 $\Pi \leftarrow pruneWorstClassifier(\Pi)$ // purge pool
11 $\Psi_k \leftarrow trainClassifier(DS_k)$
12 $\Pi \leftarrow \Psi_k$
13 $\Phi_k \leftarrow trainAnomalyDetector(L_k, \lambda)$
14 **end**
15 **end**

Algorithm 2: Prediction phase of the proposed framework

Input:

Stream - Data stream

Π - Pool of classifiers

Φ_k - Anomaly detector

Symbols:

DS_k - Data chunk

$y_{k_{pred}}$ - Predicted labels for the samples in the current chunk

Π_{Dk} - Ensemble of classifiers selected using a DES algorithm

$DSEL$ - Dynamic ensemble selection data set

1 **foreach** k, *DS* ***in*** *Stream* **do**
2 **if** $k == 0$ **then** // first data chunk
3 $DSEL \leftarrow preprocess(DS_k)$ // store DSEL for next step
4 **else** // rest of data chunks
5 $\Pi_{Dk} \leftarrow dynamicSelection(\Pi, DSEL, DS_k)$ // DES step
6 $y_{k_{pred}} \leftarrow predict(DS_k, \Pi_{Dk})$ // prediction step
7 $y_{k_{pred}} \leftarrow anomalyDetector(y_{k_{pred}}, \Phi_k)$ // refinement step
8 $DSEL \leftarrow preprocess(DS_k)$
9 **end**
10 **end**

Before providing a detailed explanation of the whole system, some general considerations must be addressed. They are provided as follows:

- For the initialization of the system, the following hyper-parameters must be selected: the size of the classifier pool (i.e., $|\pi|$, S), the chunk size (i.e., n, sample size) and the anomaly detector sample size (i.e., -1 by default, using all L_{k-1} data).
- The first data chunk (i.e., $k = 0$) is used only for training purposes and not for testing purposes. All subsequent data chunks are processed sequentially, applying first the *predictive* phase and then the *training* phase. This is a modification to the original algorithm in which distinct steps are performed for the first and second chunks.
- As all data chunks are assumed to be imbalanced, a data balancing technique (e.g., random oversampling) is applied to the learning set every time a new classifier is generated (i.e., inside the *trainClassifier ()* function of Algorithm 1).

Furthermore, as can be noticed in Algorithm 1 and Algorithm 2, the proposed solution requires the selection of a few hyper-parameters for its deployment. They are described in the following items:

- *Chunk size (n)*: the number of samples that compose each data chunk. It is a hyper-parameter that is not included in the algorithm description. However, it is assumed that all chunks have the same size and are processed in chronological order. Note that the chunk size may affect the concept drift detection capabilities of the system.
- *Classifier pool size (S)*: the number of classifiers in the pool. It may impact the concept drift adaptation capabilities of the system.
- *Anomaly detector's sample size*: the size of the data set that is used to induce the anomaly detection model. In this regard, the whole legitimate set of the new chunk can be used, a portion of it or even a bigger set by generating a cumulative data set from consecutive chunks. It may affect the accuracy of the predictions.
- *Dynamic Ensemble Selection (DES) algorithm*: the algorithm used to select the best ensemble of classifiers from the pool to predict the class of the new data. Note that the usage of distinct *DES* algorithms can provide significantly different performances, thus it is a very important hyper-parameter of the system. The *DSEL* may also have a significant impact on the prediction performance but it is not a hyper-parameter as it strictly depends on data.
- *Data set balancing method*: the technique used to balance the classes in the training set may have an impact on the detection accuracy. In this regard, different over-sampling and under-sampling techniques might be used.

Despite that the selection of hyper-parameters might have a significant impact on the performance of the detection system, the proposed solution can be implemented with some degree of flexibility, thus enabling the usage of different configurations to achieve high performance on Android malware detection in the presence of concept drift. In this regard, our experimentation evidenced that good performance metrics can be obtained with the vast majority of possible configurations. The results and hyper-parameters used in our specific implementation are provided in Section 4.3.

After setting the general considerations and describing the main variables of the proposed solutions, the following paragraphs provide a detailed explanation of the intricacies and inner workings of the proposed solution.

As specified in Algorithm 1, when the first data chunk is received (i.e., $k = 0$), the whole chunk is processed by the *splitInitialDataset()* function which takes the chunk as input, splits its n elements into S ordered and equal-sized data chunks (i.e., each composed of n/S samples), and outputs the *IDS* data set. Then, each subset is used to train a new classifier (i.e., *trainClassifier()* function) which is added to the pool (i.e., lines 4–7 of Algorithm 1). As a result, a full pool of classifiers is generated after processing the first chunk, thus available for the testing

phase of all the subsequent data chunks (i.e., *first modification* to the original algorithm). In the next step, the set of legitimate samples from the initial data chunk (i.e., all by default, but a different sample size could be used) are used to train an anomaly detection model (i.e., *trainAnomalyDetector()*). Finally, the last processing step of the initial chunk involves the storage of the whole initial chunk as the dynamic ensemble selection data set (i.e., *DSEL*) for the next chunk (i.e., line 3 of Algorithm 2). As previously explained, the *DSEL* is used to select the best classifier ensemble from the classifier pool for each data sample in the new data chunk. This selection may vary according to the dynamic ensemble selection algorithm used in the particular implementation (Ko et al., 2008).

This concludes the processing of the initial data chunk, which is used for initialization purposes, and it is the only data chunk with distinct processing steps in our proposed solution. For all subsequent data chunks, the same *first-testing-then-training* procedure is applied, described as follows.

After the first chunk is processed, when a new data chunk is received, the prediction phase is applied first, as outlined in Algorithm 2. Thus, upon the arrival of the new data chunk, the dynamic ensemble selection algorithm is fit with the previously stored *DSEL*, the classifier pool, and the new data chunk (i.e., input of *dynamicSelection()* function, line 5 of Algorithm 2). This step aims to select the best ensemble of classifiers to predict the labels for each sample in the new data chunk. Once Π_{Dk} is fit, this dynamic ensemble model is used to forecast the class of the n elements of DS_k, thus generating the initial predictions (line 6 of Algorithm 2). These initially assigned labels are then refined based on *custom* rules, included inside the *anomalyDetector()* function, and using the anomaly detector forecasts for each sample in DS_k (line 7 of Algorithm 2). The outcome of this step is the final prediction for all the samples of the new data chunk. As mentioned before, the anomaly detector helps to support or challenge the class prediction assigned by the classifier pool in borderline cases where the anomaly model may provide more accurate results. Finally, in the last step of Algorithm 2, the new data chunk is stored as *DSEL* for the next chunk.

This concludes the first processing step of the new data chunk, the *predictive* phase. The next step for the new chunk is the *training* phase, as described in Algorithm 1.

The training phase uses the whole data chunk and the outcome of the previous phase to update the pool of classifiers and generate a new anomaly detector. More specifically, the worst-performing classifier on the new data chunk is removed from the classifier pool (i.e., *pruneWorstClassifier()*). Then, a new classifier is induced using the samples from the new chunk and their predicted labels. The new classifier is added to the pool (line 12 in Algorithm 1), which is again composed of S classifiers. The removal of an *aging* classifier and the insertion of a *new* classifier keeps the pool at the specified size while updating its capabilities to accurately forecast on new data, thus being able to adapt and react to emerging concept drift. Finally, the legitimate portion of new data (i.e., L_k) is used to generate a new anomaly detector that will be used in the predictive step of the next data chunk. This last step concludes the processing of the new data chunk, using it as a training set to update the forecasting capabilities of the system.

The described *first-testing-then-training* cycle is repeated for all the subsequent data chunks in the data stream, enabling the system to address concept drift issues effectively and efficiently without major changes in the solution.

3.2.3. Concept drift characterization

The proposed solution is able to detect and adapt effectively to concept drift in Android malware detection but also provide relevant insights about its character. The characterization of concept drift can provide useful knowledge and insights about the changes in Android malware, its direction, and expectations. It can also help to enhance the trust of analysts in the detection system. The inner workings of the proposed solution can be leveraged to explore thoroughly the phenomenon of concept drift by analyzing the influence of data changes on classification quality measures in various time horizons.

For concept drift characterization and appraisal of its influence, *permutation feature importance* analysis was employed. This method, proposed by Breiman (2001), is model-agnostic and applicable to the discussed case of binary classification (i.e., malware detection) which can be evaluated by quality measures related to the classification results. The permutation feature importance technique is explained as follows.

For a matrix of feature values $\mathbf{X}$ with rows $\mathbf{x}_i$ given each of N observations and corresponding response y_i, $x_i^{\pi,j}$ is a vector achieved by randomly permuting the j-th column of $\mathbf{X}$. Given a loss function L, the importance VI_j of the j-th feature is defined as the difference between the loss calculated using pseudo-random values and the original data, as it is expressed by the following equation:

$$VI_j^{\pi} = \frac{1}{N}\sum_{i=1}^{N} L\left(y_i, f\left(x_i^{\pi,j}\right)\right) - L(y_i, x_i) \tag{1}$$

It is worth mentioning that Random Forest algorithm offers an alternative assessment based on *Gini coefficient* or *entropy* (Maimon & Rokach, 2005), called *feature importance*. However, such calculated importance is based on the training data used to create the classification model. In the discussed case, a more important issue is how the model works on new data, which might be affected by concept drift, rather than how well the features were used to discriminate the learning set. Therefore, the application of the permutation feature importance technique to the test data is preferred. In this regard, due to its stochastic nature, the permutation feature importance score may vary significantly among iterations. Hence, for the sake of results' stability, it is recommended to repeat the permutation procedure at least 50 times and average the results (Altmann et al., 2010).

The concept drift characterization method used in this study adopts Eq. (1) by the creation of the classification function f_t using data X_t from period P_t. Next, observations X are taken from the set $\bigcup_{l=t+1}^{l+h} X_j$ where h declares an analysis time horizon. In this work, we discuss the following time horizons: *short-term* (i.e., 3 months), *mid-term* (i.e., 6 months) and *long-term* (i.e., 12 months). The usage of several time horizons brings an opportunity for better characterization of the changes in the importance of features. The whole procedure is summarized by the following equation:

$$VI_j^{\pi}(t) = \frac{1}{N}\sum_{\substack{i=1,\\ x_i \in \cup_{l=t+1}^{l+h} X_l}}^{N} L\left(y_i, f_t\left(x_i^{\pi,j}\right)\right) - L(y_i, x_i) \tag{2}$$

The procedure can be used to evaluate the influence of features on various quality functions $Q(.) = 1 - L(.)$ such as:

- *F1 score*, which is a more comprehensive metric for malware detection performance on imbalanced data sets than the overall accuracy and it is defined as:

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{3}$$

- *Specificity (True Negative Rate)*, which describes the quality of benign software recognition (i.e., negative label). It is calculated as:

$$TNR = \frac{TN}{TN + FP} \tag{4}$$

- *Recall (True Positive Rate)*, which describes the quality of malware detection (i.e., positive label) and it is defined as:

$$TPR = \frac{TP}{TP + FN} \quad (5)$$

where *TP* (i.e., true positive) refers to the number of correctly recognized malware among all test instances. *TN* (i.e., true negative) reflects the number of correctly recognized benign software among all test data. *FP* (i.e., false positive) provides the number of instances incorrectly predicted as malware among all test samples, and *FN* (i.e., false negative) the number of incorrectly predicted samples as benign data in the test set.

The analysis of permutation feature importance scores of chronologically ordered data chunks allows the exploration of changes and the observation of the evolution of important features in the data, which enable the detection of trends and the characterization of emerging concept drift.

4. Results

4.1. Data pre-processing

After the application of each sequential data pre-processing step, the results reported in Table 1 were obtained.

As can be observed, after the first pre-processing step, 128 syscalls were found to be non-zero and not constant valued. This filtered subset of features was further processed and highly correlated features (i.e., $|r| > 0.80$) were removed. More specifically, 31 features were found to be over the specified threshold, thus showing a *strong* linear correlation with another feature. The resulting feature set was composed of 97 features.

To assess the adherence to the normal distribution of the feature distributions, four normality tests were applied to every feature, including the *Shapiro-Wilk* normality test, the most powerful normality test according to Mohd Razali & Bee Wah (2011). The results of the statistical analysis proved that no feature showed Gaussian distribution characteristics, as evidenced graphically by the plots of feature distributions in Fig. 2. Therefore, the final feature set was composed of 97 non-normally distributed syscalls.

4.2. Concept drift detection

The initial period selected for concept drift detection was the second semester of 2011. As this study compares the performance of two timestamps for concept drift detection and handling, it was critical to select a period where effective models could be induced for both timestamps. Thus, this period was preferred as it provided enough data to build effective classification models for both timestamps (i.e., *accuracy* > 95%). Data from prior periods were discarded and not further analyzed. Random Forest classification models were induced using the whole feature set (i.e., 97 syscalls) and the class labels of the samples corresponding to each timestamp.

The most relevant features of the classification models were selected using the permutation feature importance technique (i.e., feature selection). In our experimentation, 500 permutations per feature were used, which is significantly over the empirically recommended quantity for results' stability (Altmann et al., 2010). From the obtained results, only the features that showed positive mean importance on the initial-period model for each timestamp were selected and ranked in descending order of importance. The following results were obtained for each timestamp:

- *Last modification* timestamp: 32 features were found *important* from the whole feature set (i.e., 97). This selected feature set is referenced as *initial-lm-set*. The data set for this period was composed of 9,288 samples (i.e., 6,916 legitimate apps and 2,372 malware apps), and the accuracy of the RF classification model on the testing set was 0.9870.
- *First seen* timestamp: 17 features were found *important* from the whole feature set. This selected feature set is referenced as *initial-fs-set*. The data set for this period was composed of 2,677 samples (i.e., 2,124 legitimate apps and 553 malware apps), and the accuracy of the RF classification model on the testing set was 0.9859.

After feature selection, the resulting feature sets were used to build one-class anomaly detection models using the Isolation Forest algorithm (i.e., 300 estimators per model). As a result, for each class (i.e., malware and benign) in each timestamp-related data set, a one-class anomaly model was generated, using the corresponding feature set as model features (i.e., *initial-hm-set* or *initial-fs-set*). Then, the malware and benign data belonging to *posterior* time frames were split into 6 months periods (i.e., from 2012 to 2020) and used as testing sets for each corresponding timestamp-class model. Besides, for every timestamp-class combination, 3 anomaly models were induced using distinct subsets of features from the *important* feature sets (i.e., best 5 features, best 10 features, and all features).

The accuracy metrics of all the induced anomaly models on their respective testing sets were retrieved. The results are provided in Fig. 3. The line graph on the left shows the results related to the *last modification* timestamp, while the line graph on the right provides the data related to the *first seen* timestamp. The 6 anomaly models generated for each timestamp are reported with different colors and line styles. The color reflects the data class (i.e., red for malware and green for benign apps). The line style informs about the subset of features that was used to build and test each specific anomaly model. More precisely, solid line is used for all features, dashed line for the 10 *most important* features, and dotted line for the 5 *most important* features. The horizontal axes provide the test period, whereas the vertical axes provide the *accuracy* value retrieved for each specific period. The horizontal axes are split into 6-months periods. The *.1* value attached to the year number informs about data belonging to the first semester of that year (e.g., 2012.1), whereas *.2* reflects the data regarding the second semester (e.g., 2012.2). As a result, six anomaly detection models were built and tested per timestamp (i.e., three per class) encompassing the whole 2012–2020 time frame.

The anomaly detection results provided in Fig. 3 demonstrate the existence of concept drift in the data, thus proving that the same set of features and values are not useful in all time frames to recognize either one of the classes. In this regard, according to the concept drift typology proposed by Ramirez-Gallego et al. (2017), the following behaviors are observed.

In benign applications, an *incremental drift* dominates. The number of recognized observations slightly goes down over time to dip in the last period in a *sudden drift*. However, this deep dip in performance might have been caused by the scarcity of samples available for this last period in the data set (e.g., over 1000 are available for 2020.1, whereas less than 40 for period 2020.2). Thus, except for the last period, the observed behavior is typical for an AI system with a static learning set tested on data evolving over time, as in Luckner (2019).

The data *drift* is especially evident in malware data. The graphs depict that the initial model, trained on any feature set, shows remarkably distinct accuracy values from period to period, suggesting that the importance of features for the classification models might have changed significantly, thus evidencing concept drift. Both timestamps provide a similar picture of the phenomenon, with the initial models performing well on data belonging to closer periods and losing discriminatory power over time. In 2016.1 and 2019.1, the initial set of important features seems to become relevant again, reaching accuracy levels similar to benign data, but losing its importance in the subsequent periods, thus leading again to data *drift* and poor discrimination. It could be related to a *recurrent* threat emerging in the initial, 2016.1 and 2019.1 periods.

In any case, the analysis of the line graphs in Fig. 3 evidences the presence of concept drift in Android behavioral data, which is especially pronounced in the malware case. In consequence, to build long-lasting

and robust Android malware detection solutions, the detection systems must be able to adapt and learn from the changes in the data to keep high and stable performance over time.

4.3. Concept drift handling

The proposed solution, detailed in Section 3.2.2, was applied to KronoDroid data set, described in Section 3.1. As the data set is not a real data stream and encompasses a long period, the data was divided into time-constrained data chunks. This operation enabled us to simulate a realistic scenario where the data flow is constant and in a great volume. In this regard, the samples in each data chunk are likely to be similar until *drifting* emerges in any of its forms. Besides, as the 6 months periods used in Section 4.2 might be too wide to accurately detect emerging concept drift, a shorter temporal constraint was established. Thus, a maximum of 3 months of data were included in every chunk (i.e., referenced as year quarter, Q). In addition, to train the models, a fixed chunk size of 4000 samples was established. In the case that the incoming data for a specific time frame did not contain enough data to cover the 4000 samples per chunk of the training phase, prior data were used respecting the chronological order.

The data set used provided a large number of samples to cover the years from 2011 to 2018, but there were not enough timestamped data to cover effectively the requirement of 4000 samples per quarter in the years before and after this time frame (i.e., 2008–2010 and 2019–2020). Consequently, just the data from 2011.Q3 to 2018.Q2 were used in the experimental setup, thus covering 7 full years of Android history.

Based on experimental testing, the values of the hyper-parameters selected for our implementation were: 4000 samples per training data chunk, pool size of 12 classifiers, and *Random Forest* models with 300 estimators as classification models. The dynamic ensemble selection algorithm used was *META-DES* (Cruz et al., 2015). *Isolation Forest* algorithm was used to induce the anomaly detection models. The data balancing technique used was *random oversampling*.

Although no hyper-parameter optimization procedure was performed, the selected hyper-parameters provided high and stable performance. A different selection of hyper-parameters may yield similar performance metrics as the solution is robust and allows certain degree of flexibility in the selection of the hyper-parameters. The implementation was coded in Python programming language, leveraging the functionality of *scikit learn*, *imblearn* and *deslib* libraries.

The performance of the proposed solution when the *last modification* timestamp data was used is reported in Fig. 4, while Fig. 5 provides the performance for the *first seen* timestamp data.

Fig. 4 provides the F1 score performance of the proposed solution, using the provided hyper-parameters, and its comparison with two naive solutions and the original algorithm. More precisely, the *initial classifier* line (i.e., dotted grey line) provides the performance results of a classifier generated using the data of the first chunk and tested on all the subsequent chunks. This approach simulates the scenario where a detection model is generated at a specific time (i.e., 3rd quarter of 2011 in this case) and is never updated, thus neglecting concept drift. As the initial data chunk was significantly imbalanced towards the legitimate class (i.e., 98% of the data points were benign apps), another *naive solution* is provided as a reference (i.e., dashed grey line), using data from the second chunk, where the data were more balanced (i.e., 65% legitimate, 35% malware). As can be observed, the two naive solutions, which are never updated, yield poor detection performance as time passes. On the contrary, the proposed solution (i.e., solid blue line) provided a detection performance of over 90% in almost all periods, showing robustness against concept drift, reacting, and updating its knowledge when it emerged. Further, its performance was superior to the performance shown by the original algorithm in most periods, especially in the first ones.

As can be seen in Fig. 5, when the *first seen* timestamp is used, the high, stable, and smooth performance line provided by the solution in the previous case is not observed. The performance line performs sudden dips and boosts that might have been caused by a general temporal misplacement of the data which led to improper concept drift handling. This timestamp does not seem as reliable as the *last modification* timestamp to locate the data samples in their correct period and, consequently, the changes in data features do not emerge naturally but artificially, likely caused by temporal displacement. Despite that, the solution still shows good performance and adaptation over time.

It is worth noticing that, in this case, the horizontal axis starts and ends a period later than in Fig. 4. This difference is due to the distinct temporal

Table 1
Data pre-processing results.

Preprocessing step	Results
Feature Variance Analysis	160 constant or zero-valued
Feature Correlation Analysis	31 highly correlated
Feature Distribution Analysis	0 normal
Final feature set	97 non-normal syscalls

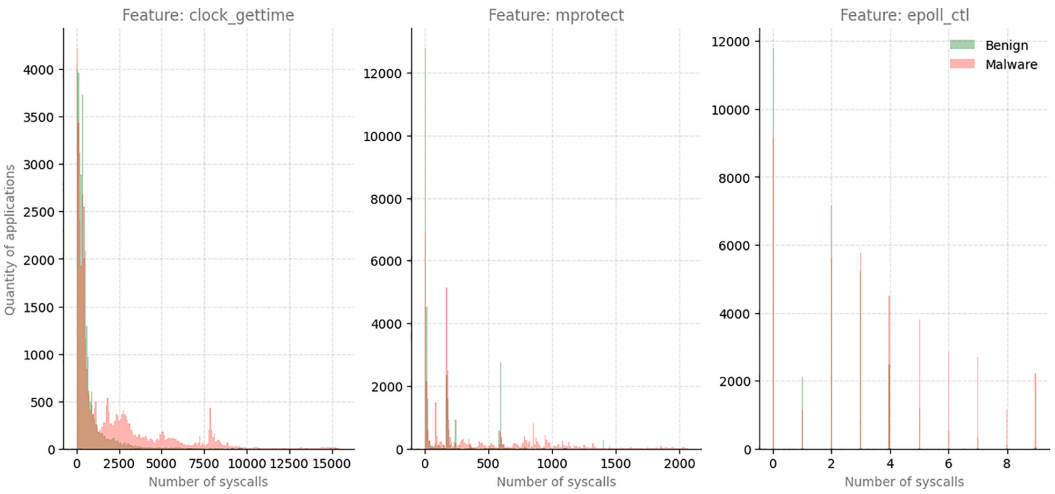


Fig. 2. Distributions of example features.

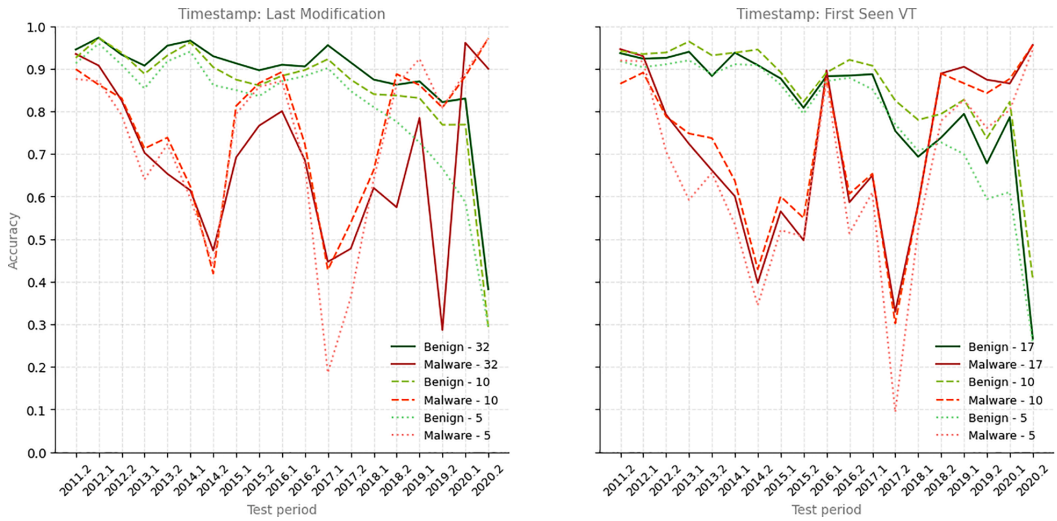


Fig. 3. Performance of the one-class anomaly detection models.

distribution generated by this timestamp, which allowed generating the initial classifier on 2011.Q3 but not a test set for that period. Therefore, the time series data is displaced one quarter with respect to Fig. 4. This fact supports the aforementioned differences between these two timestamps regarding the location of data samples within the historical timeline.

As can be observed in Fig. 5, when the performance of the proposed solution is compared with the other approaches, it significantly outperforms the *initial classifier* and the *naive solution*, reaching high-performance values in almost all periods. Again, it outmatches the performance of the original algorithm, especially in the first periods. Despite that, when the *first seen* timestamp is used, the performance of the system is less smooth and remarkably lower than when the *last modification* timestamp is used. More precisely, the displacement of data samples over the historical timeline overrides the emergence of a *natural* concept drift, thus hindering its proper handling.

In conclusion, the usage of single period classifiers, applied over time with no update, proved to be inefficient and showed poor and degenerative performance as time passed. These solutions become obsolete and ineffective in a short time span. Contrarily, the proposed solution is robust and resilient to changes in data over time, especially when the *last modification* timestamp is used, keeping high-performance metrics on Android malware detection under the challenge of constant data evolution. These results also demonstrate that system calls can be used to achieve effective, long-lasting, and robust Android malware detection even when concept drift threatens the performance of the detection system.

4.4. Concept drift characterization

The following paragraphs explore the concept drift phenomenon using several instruments for its characterization. More specifically, the impact of the *pool size* and the *evolution of the importance of features* are analyzed.

4.4.1. Impact of the pool size

The analysis of the proposed solution according to the number of classifiers present in the pool of classifiers brings some interesting findings on the concept drift analysis and the modeled data.

Different pool sizes for the proposed solution, ranging from 2 to 20 classifiers, were assessed. The experiment was repeated 20 times per pool size and the results were averaged. In this regard, Fig. 6 provides relevant information regarding the experimental results such as the *average lifetime of a classifier* (i.e., how long, on average, a classifier was in the pool before it was removed), the *average lifetime of the classifiers from the initial pool* (i.e., in what period, on average, all the classifiers created in the first period were removed), and the *quality of the new classifiers* (i.e., how many times, on average, the most recent classifier was the best performer) calculated for both timestamps. The color ribbons surrounding the lines in Fig. 6 provide the standard deviation for each reported value.

As can be observed, regardless of the timestamp, the average lifetime of a classifier inside the pool is linear according to the pool size in a ratio of 0.8–0.9. This fact shows that the *oldest* classifier in the pool is not always the one removed when the poorest model is purged from the pool. Recurring threats might cause old classifiers to perform relatively well in later periods. Furthermore, the results show that regardless of the pool size, a single classifier is never the best performer for more than 5 periods, thus demonstrating the dynamism of the phenomenon.

More interestingly, the number of periods in which the newest classifier is the best slightly decreases when the pool size increases. As can be observed, in all models the number of times a classifier is the best performer is different from 1. This shows that there are periods where an older model is repeatedly the best performer and suggests that there might be gaps between periods where the same classifier is the best performer, thus reinforcing the existence of concept drift in the analyzed data. Besides, it should be noted that newly created classifiers are valid for a longer time for the *first seen* timestamp than for the *last modification* timestamp. Hence, *natural* concept drift handling seems to generate more specific and better classifiers with reduced lifetimes, as shown in Fig. 4, whereas misplaced data generate more generic and worse performer classifiers, but with longer lifetimes, as displayed in Fig. 5.

In the case of the *last modification* timestamp, the classifiers from the initial pool are always removed as soon as possible (i.e., in the first S periods, where S refers to the pool size). This shows that the initial data cannot be used effectively to discriminate new data, so the related classifiers are rapidly removed. This observation concurs with the results obtained on the anomaly detection models (see Fig. 3). For the *first seen* timestamp, the lifetime of the initial pool substantially increases for seven and more pool components. This suggests that the knowledge from the initial periods is useful for later periods, which might be caused by a general misplacement of the data samples along the timeline, provoking *artificial* drift in the data.

Although this thorough analysis yielded relevant insights about the

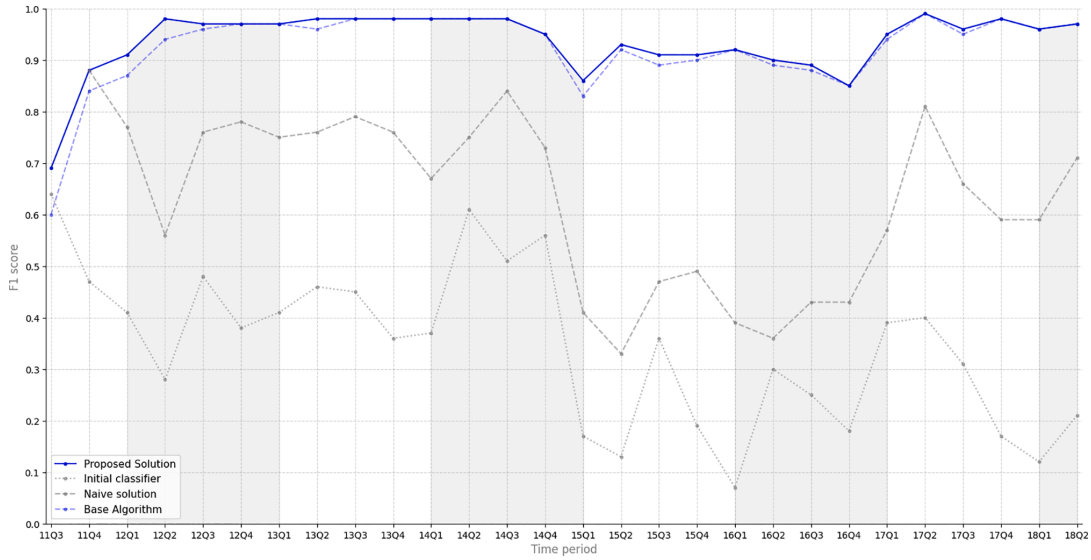


Fig. 4. Proposed solution performance using the *last modification* timestamp.

impact of the pool size on concept drift and data modeling, it did not provide any hint about the optimal pool size, a relevant question that remains unsolved. The following paragraphs address this issue.

Fig. 7 shows the classification results (i.e., F1 score) obtained using the proposed solution with pools of various sizes. The usage of boxplots enables us to easily compare the distributions of the classification results for distinct pool sizes. The dark blue boxplots provide the results for the *last modification* timestamp while the light blue boxplots provide the distributions of the classification results for the *first seen* timestamp. The body of the boxplots reflects the range where the central 50% of data is located, also referenced as *interquartile range* (*IQR*). The IQR is calculated as $IQR = Q3 - Q1$, where $Q3$ and $Q1$ are the 75% and 25% percentiles, respectively (i.e., the borders of the box). The orange line crossing the bodies references the median while the average is provided by the red rhombus. *Outliers* or extreme values are reported as grey dots, located further than the minimum and maximum *whiskers* which are calculated as $minimum = Q1 - 1.5 * IQR$ and $maximum = Q3 + 1.5 * IQR$.

The boxplots in Fig. 7 reflect that even though the average quality diminishes for larger pools in both timestamps, in general, the pool size does not seem to influence substantially the classification quality. However, the pool size of 12 classifiers for the *last modification* timestamp shows distinctive properties. First, the average and median values are nearly the same, defining a relatively *symmetrical* distribution. Thus, the deviations of all terms from the median cancel out. This distribution is different from the other distributions, which are skewed, thus making the median a better central measure than the average and relating to the existence of extreme values. Second, the number of outliers in the distribution is minimal. The other pool sizes generated classifiers with

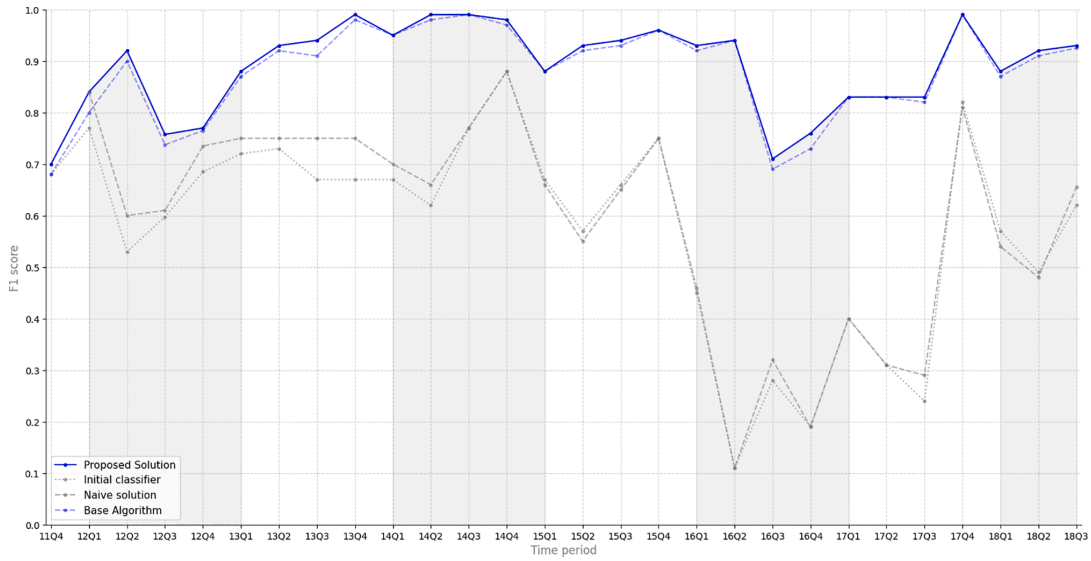


Fig. 5. Proposed solution performance using the *first seen* timestamp.

worse and more spread results. Third, even though the pool size of 13 shows similar properties, the whiskers for the pool size of 12 are shorter, thus concentrating the data in a shorter range, with less variability. Besides, as a smaller number of classifiers is needed, it is a more efficient approach than the pool size of 13 components. As a result, all these observations make the pool size of 12 the preferred option for optimal results.

As can be seen in Fig. 7, in the case of the *first seen* timestamp, the obtained results for all pool sizes are remarkably worse than for the *last modification* timestamp.

The combination of these results with the previous findings from Fig. 6 enables us to confirm that the system relied excessively on old classifiers when using the *first seen* timestamp. Two aspects that stress it are the delay in replacing the initial pool and the extended lifetime of the classifiers. Due to the discretionary nature of the generation of the *first seen* timestamp, which depends on users proactive submissions, it seems to lag behind the *last modification* timestamp unpredictably, thus being prone to displace the samples within the historical timeline and provide a relatively inaccurate temporal location. Therefore, the obtained results suggest that the *first seen* timestamp provides a less realistic approximation to the *real* concept drift and, consequently, a less accurate solution and characterization of it.

Based on these results, the following experiments were performed using the pool size of 12 components and the *last modification* timestamp.

4.4.2. Evolution of features importance

The permutation feature importance technique was used to analyze the evolution of the importance of features over time, thus enabling to characterize concept drift. The procedure and main results of this stage are explained as follows.

For each period P_i, the best classifier was selected according to the results obtained in Section 4.3 and detailed in Fig. 4. The permutation feature importance technique applied to the classifier was calculated using Eq. (2) with F1 score as loss function. The *importance* was calculated separately for three test sets (i.e., time horizons). The first set was the subsequent period to P_i, thus P_{i+1}. The second set consisted of two successive periods, $\cup_{j=i+1}^{i+2} P_j$, and the third set contained the four subsequent periods, $\cup_{j=i+1}^{i+4} P_j$. As defined, the sets were built incrementally, thus corresponding to three, six, and twelve months data horizons. The results were calculated for all possible periods of the data set in the range $P_1,\ldots,P_{n-4}$.

The usage of three incremental test sets enabled us to observe how the importance of features varied in short, medium, and long-term time horizons. In this regard, Fig. 8 provides the distributions of feature importance using boxplots, calculated for all periods and including all syscalls that reported a non-null importance in at least one period. The box color indicates the time span or horizon (i.e., darker colors reference longer time-frames). The orange line crossing the body indicates the median and the green triangle provides the average value. The horizontal axis informs about the system call name, while the vertical axis reflects increasing scores of permutation feature importance (i.e., a larger score directly relates to greater importance).

The results provided in Fig. 8 were obtained from 20 tests of 500 permutations each. Even though the results slightly varied among iterations, the main findings described in the following analysis were common for all tests.

As can be observed in Fig. 8, no feature was found useful or *important* in all tests as all boxplots start near the zero value. A fact that stresses the existence of concept drift. More interestingly, based on these results three types of features can be distinguished. The first type of features includes those features that are not useful in any time horizon like *getgid32* or *restart_syscall*. These features might have provided a low importance score in some periods due to the stochastic nature of the technique or a non-random positive importance but with a negligible impact on the task. The second group of features is related to features that are more important in longer time frames (i.e., medium and long term) than in the short-term. These features are not very good at recognizing sporadic threats, but they constitute a solid base in a time-extended threat detection system. Features like *clock_gettime* and *flock*, which lie inside this category, show a relatively stable discriminatory power over time. Lastly, the third type of feature shows the opposite situation, the feature is a relatively good discriminator in the short term but is not as useful in longer time frames. Due to the larger number of distinct threats present in longer time frames (i.e., more families and malware variants), these features are not so useful for overall discrimination as in the short time frame, where a smaller variety of threats is present. Consequently, these features might work well to distinguish specific malware families. Features such as *write* or *SYS_317* are included in this category.

To perform a deeper analysis of the importance of features for specific recognition tasks, the permutation feature importance was calculated using *specificity* and *recall* as loss functions. The results for

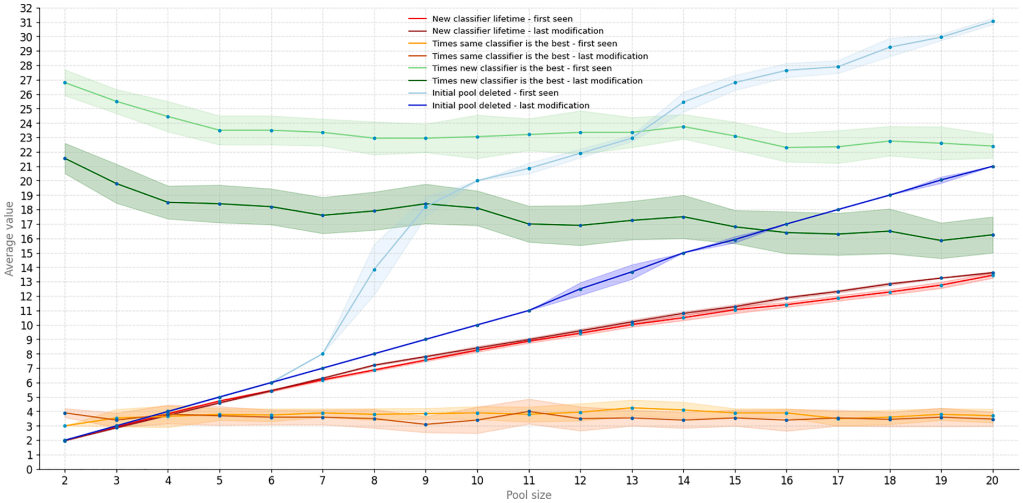


Fig. 6. Classifier pool statistics for both timestamps.

specificity provide information about important features to recognize benign software, whereas for *recall*, also called *sensitivity*, they describe important features for the malware detection task.

The obtained results are depicted in Fig. 9, showing features with positive importance in benign and malicious software recognition tasks and their evolution over time. The presentation is limited to features with positive mean importance estimation obtained using Eq. (2) and that, for each task, were found in the top 3 most important features in any period. The horizontal axis provides the timeline, split into quarters or periods. Regarding the vertical axis, the color relates to specific features, while the colored areas (i.e., vertical range) in each stacked bar provide the importance score of the specific features relative to the total importance of each specific period of time (i.e., the total importance of a period is the sum of the importance scores of all the important features in that period). Consequently, the larger the vertical range or area spanned by a feature in a bar, the greater the importance of the feature in the period.

In the case of benign software recognition, presented in Fig. 9a, the importance of features appears to be locally stable. Several features like *read* and *mprotect* have similar influence for extended periods of time (i.e., from 2011-Q4 to 2014-Q2 in the case of *read* and from 2012-Q1 to 2017-Q1 in the case of *mprotect*). Besides, quarters with clearly outstanding features are rare (e.g., 2011-Q4, 2017-Q2). So, despite that some trends can be spotted, with some features gaining and others losing importance in some periods of time (e.g., *SYS_310* increases from 2016-Q1 to 2017-Q1 and *flock* increases from 2016-Q1 to 2018-Q1), the overall picture shows stability and that the same set of features is relevant in all time frames with no distinctive changes in relative influence and no new *important* features emerging over time (i.e., the bars have either a small portion of grey area or no grey area, meaning that most of the important features for each period are included in the bars).

The results are drastically different for the malware recognition task. Fig. 9b shows the changes in feature importance calculated for the recall function. As can be noticed, for most quarters, the dependencies observed in a specific period are not repeated in the following periods. Besides, even when a feature shows an extremely high importance in one period (e.g., *pread* in 2014-Q2), no consistency is observed and the importance of the feature dramatically decreases in the next periods. The only remarkable exception is *clock_gettime* feature, which is a very important discriminatory variable for several years (i.e., from 2012-Q3 to 2015-Q3). However, even in this case, there are quarters in this extended time frame where the feature loses completely its discriminatory power for malware detection (i.e., from 2014-Q2 to 2015-Q1).

Based on these observations, it is worth analyzing how two relevant features, *clock_gettime* and *pread*, were represented in the time horizon analysis performed previously. In this regard, Fig. 8 shows that *clock_gettime*, the feature found important for an extended period of time, is more important for the medium and long-term time horizons than in the short-term. In contrast, *pread*, the feature that was found critically important for a single period, obtains similar results in all horizons. Therefore, the horizon analysis supports the previous observations. In any case, it should be stressed that any importance score (i.e., local or periodical) influences all three horizons, but that the relationship among the levels gives additional information about the character of the importance.

Another issue observed in the malware recognition case is the existence of periods where the total importance of the features included in the bar is far from reaching the top (i.e., 2014-Q4, 2018-Q2). In those periods, none (e.g., 2014-Q4) or few of the included features (e.g., 2018-Q2) were found important for the malware recognition task. In the former case, it suggests that the set of features was not large enough to model all malware types observed in the data, whereas in the latter case, new features, not important in other periods, emerged as important.

Finally, even though important features seem to vary dramatically among quarters for the malware recognition task, some general patterns can be spotted. For instance, as mentioned before, *clock_gettime* is critically important from 2012-Q2 to 2015-Q2 but not so much after (i.e., more recent years). The internet-related system calls (i.e., *socketpair*, *recvfrom*, *setsockopt* and *getsockopt*) appear to have more importance in the recent years, from 2015-Q4 to 2017-Q3. More interestingly, the bars from 2012-Q1 to 2016-Q1 show clear dominance of small subsets of features (i.e., mainly *clock_gettime*), whereas in the latter years, the bars are composed of more features, looking more similar to the bars of the benign recognition task. In this regard, it is worth noting that, when comparing Fig. 9a with Fig. 9b, the segmentation of the bars is a major difference between them. Specifically, for the benign recognition task, the bars are dense, composed of many features, and show stability (i.e., the same set of features shows similar importance over years). On the contrary, the bars for the malware recognition task are mostly composed of a small subset of features, showing clear dominance of some of them

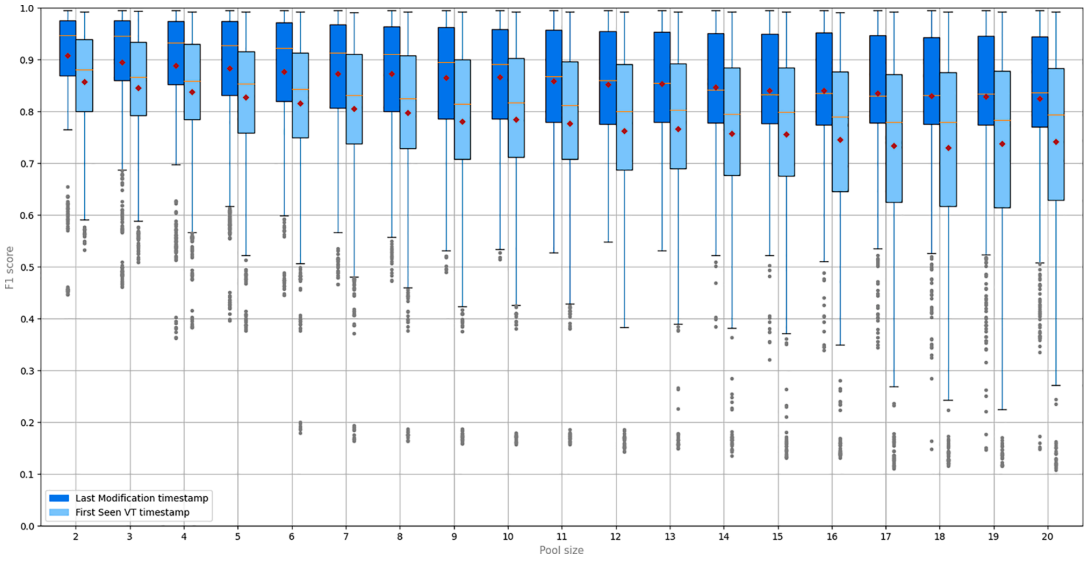


Fig. 7. Classification performance boxplots for both timestamps.

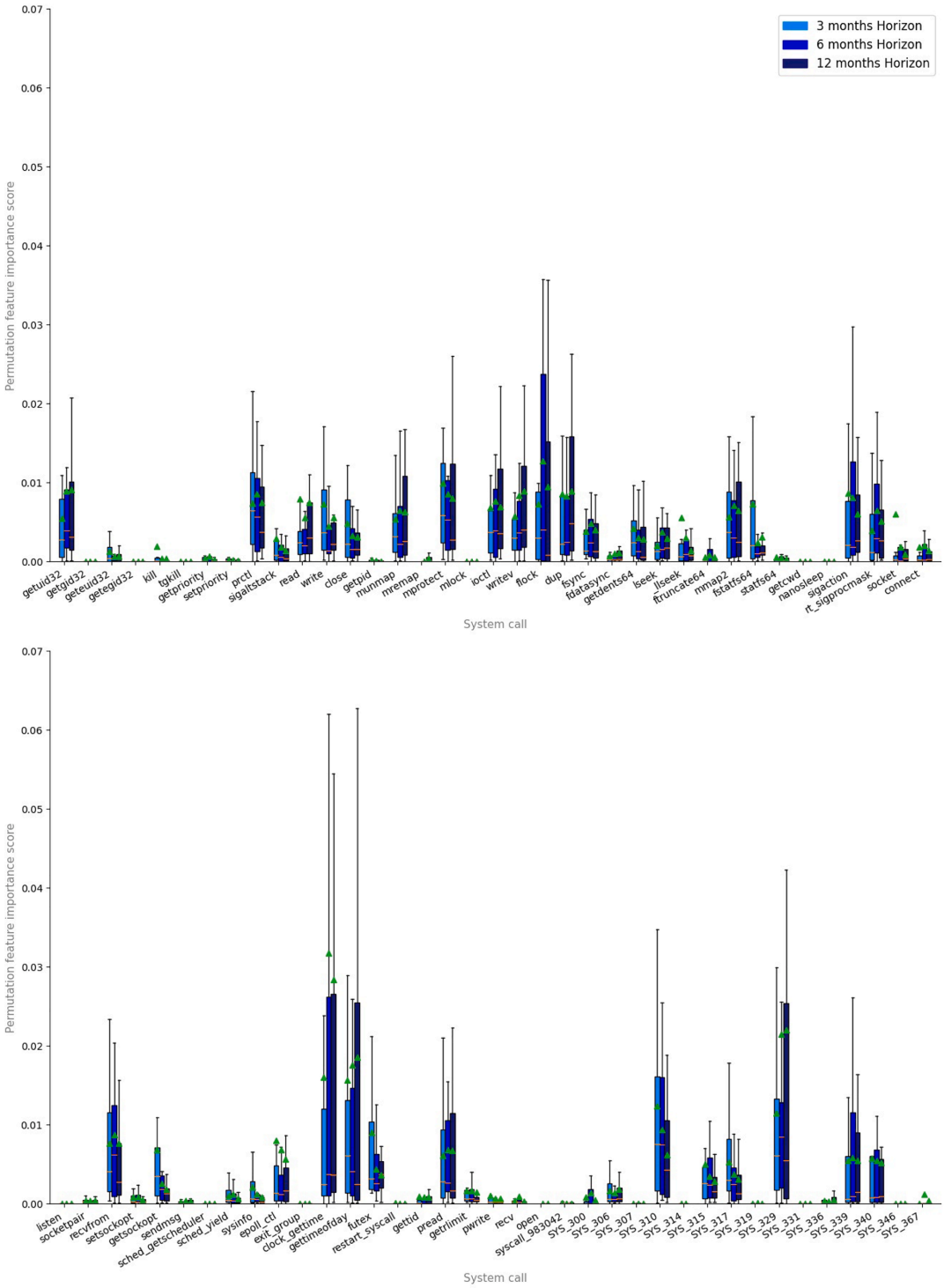
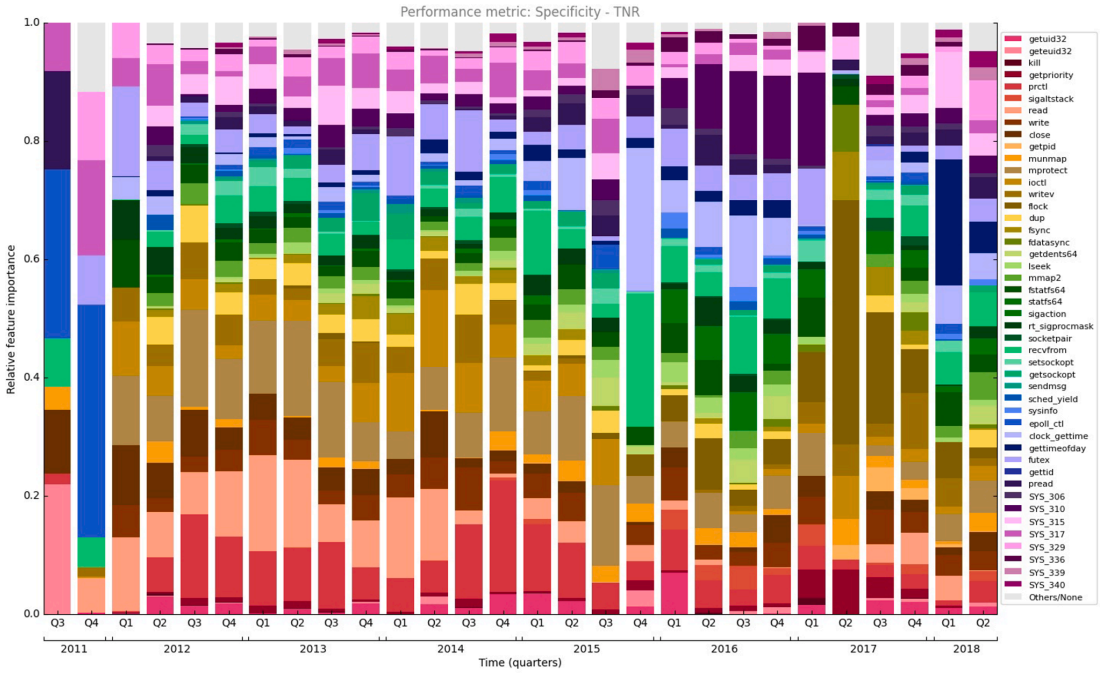
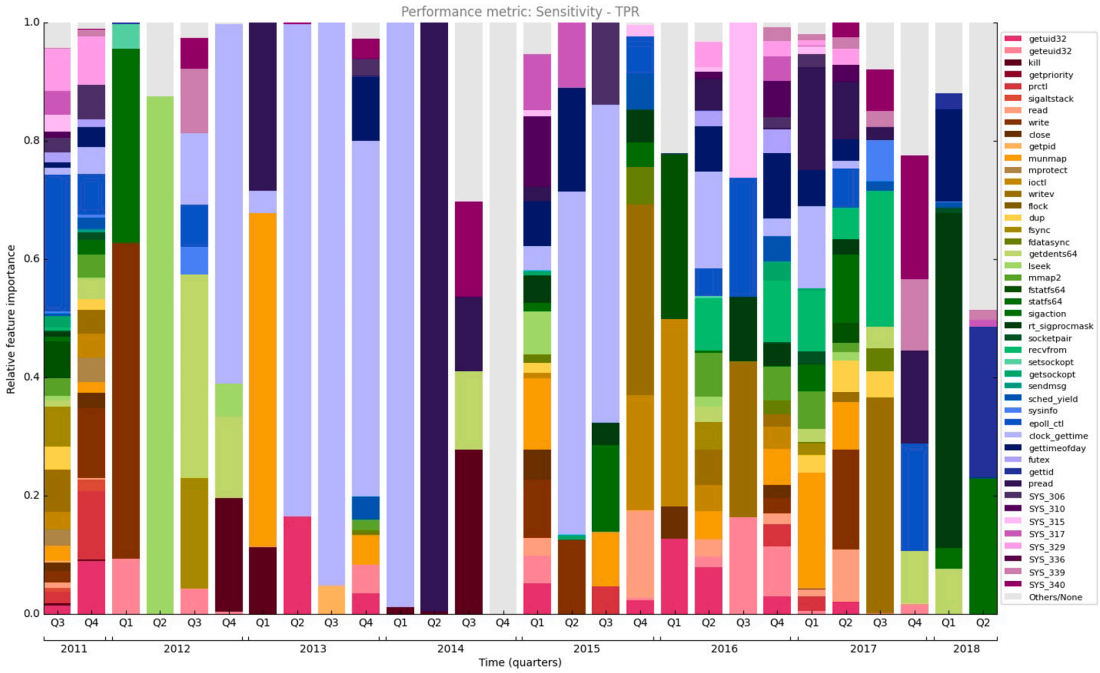


Fig. 8. Boxplots providing feature importance distributions calculated for *short*, *medium* and *long-term* time horizons.



(a) Specificity



(b) Recall

Fig. 9. Feature importance calculated quarterly for specificity (a) and recall (b) tasks.

over the rest. Consequently, the malware recognition task appears to be significantly more complex and changing more rapidly than the benign software recognition task.

To sum up, in this section we performed a thorough analysis of the evolution of the importance of relevant features. This analysis provides relevant insights about the data evolution, assisting to characterize concept drift which, in turn, can help to understand malware changes and their direction to design and implement better detection systems.

5. Discussion

The phenomenon of concept drift in Android malware detection has been neglected by most of the specialized research in the domain, which has overlooked the degenerative impact of *time* in the machine learning-based malware detection systems. The reduced number of studies that considered the impact of time in their detection systems proposed solutions to address the issue mainly based on static approaches (i.e., API calls) and did not provide any characterization of the phenomenon.

To the best of our knowledge, the solution proposed in this study is the first to tackle the concept drift issue in Android malware using dynamic features and achieving long-term high performance. The previous solutions focused on static features (i.e., API calls) and spanned shorter time frames. In this regard, *DroidEvolver* (Xu et al., 2019) used API calls as features and obtained significant results in a 6 year-long time frame (i.e., 2011–2016), outperforming *MaMaDroid* (Onwuzurike et al., 2019), a prior solution. As shown in Fig. 10, the solution proposed in this study outperforms the state-of-the-art solution, *DroidEvolver*, both in time and detection performance (i.e., F1 score). More specifically, when the training period is excluded (i.e., 2011), our proposed solution achieves an average F1 score of 94.05% in the 2012–2016 time frame, whereas *DroidEvolver* averages 89.56% in the same period of time. When the longest period is considered, from 2012 to 2018, our proposed solution's average F1 score increases to 94.65%. To assess the statistical significance of the difference between both solutions, Wilcoxon's signed-rank test for paired scores (Japkowicz & Shah, 2011) was used. The results confirmed that our solution, in the time frame from 2012 to 2016, performs significantly better than DroidEvolver at the confidence level of 0.05 ($p\text{-}value = 0.048$). However, as our solution could not be trained with full 2011 data (i.e., just samples from Q3 and Q4), the statistical significance of the results could not be confirmed at confidence level 0.05 ($p\text{-}value = 0.197$) when the training period was included (i.e., 2011–2016).

Besides, is it worth noting that the data features used to build our approach are distinct from the ones in the related literature, including *DroidEvolver*. *DroidEvolver*, as most related solutions, uses API calls (i.e., static features), whereas our proposed solution uses system calls (i.e., dynamic features). In addition, none of the previous studies that dealt with Android concept drift provided any characterization of it, which hinders the interpretability of the results and the comprehension of the phenomenon. The solution proposed in this study has been proved effective to address concept drift in Android malware detection and characterize it.

The related solutions focused on F1 score performance, not providing any other performance metric. As a result, the comparison between solutions is restricted to the F1 score metric. For the sake of completeness and better comparison of other solutions with this work, a summary of other relevant performance metrics of our proposed solution is provided as follows. The proposed solution averaged 95.17% precision, 94.14% recall, and 89.49% specificity in the 2012–2018 time frame. These metrics emphasize the goodness of the proposed solution to effectively tackle concept drift while keeping high-performance metrics for the whole study period.

A distinctive point of this study is the evaluation of distinct timestamps to date the apps and the assessment of their impact on concept drift detection and handling. The *KronoDroid* data set enabled the usage of distinct timestamps on our evaluation, thus providing results based on relevant timestamps (i.e., *last modification* and *first seen*). To the best of our knowledge, no previous study in the field has evaluated distinct timestamps for concept drift detection and handling. More precisely, the concept drift-related studies in the literature do not usually provide details about the used timestamp or justify the usage of a specific approach. But, if they do, they do not assess the reliability of the timestamp. In this study, we address such issues by providing and comparing two relevant and useful timestamps for Android malware detection concept drift handling. The systematic usage of an *internal* timestamp (i.e., *last modification*) rather than *external* timestamps (e.g., *first seen*) has proved to be reliable and accurate to handle and characterize the phenomenon. Besides, the usage of the last modification timestamp may help to avoid errors and data misplacement caused by human-related techniques (e.g., user submission delay), thus enhancing historical accuracy. As shown in Section 4.3, the proposed solution using the *last modification* timestamp proved to be more accurate and reliable than when the *first seen* timestamp was used to locate applications in the Android historical timeline, thus generating a more effective detection solution.

The features used in this study (i.e., system calls) have demonstrated great effectiveness and consistency to deal with concept drift. In this regard, just a small subset of the whole feature set was used to build an

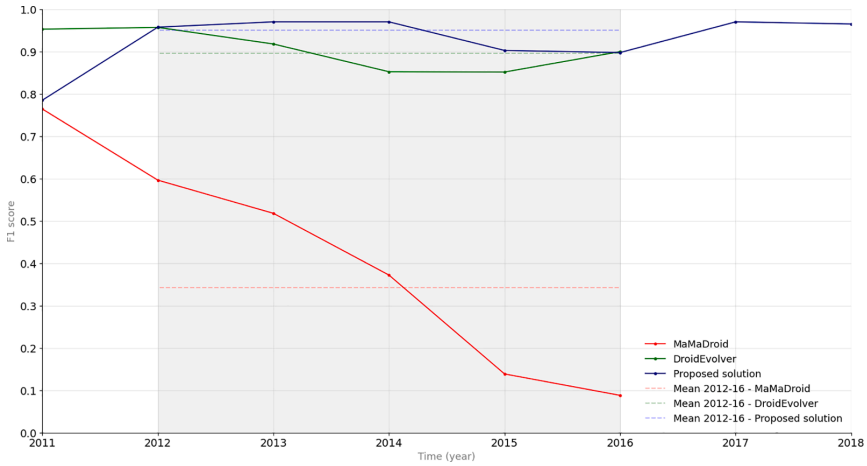


Fig. 10. Comparative performance of the proposed solution with the state-of-the-art solutions.

effective solution (i.e., 97 features). The system calls feature set is consistent, as system calls are rarely modified at kernel level which provides long-time stability for the feature set. This is radically different from the changing nature of API calls, which are prone to suffer constant modifications and the addition of new features on every new Android framework release. This constant modification of the features generates the need to constantly update the feature set to keep the models updated, which may end up in an increasingly large, ever-growing feature set. For instance, in *DroidEvolver*, the API calls feature set grew from 14,327 features in 2011 to 52,001 features in 2016 (Xu et al., 2019). On the contrary, our solution kept the same feature set constant, composed of just 97 features, for a longer period of time (i.e., 2011 to 2018). Furthermore, the usage of highdimensional feature sets may harm the performance of machine learning-based detection systems, a phenomenon called the *curse of dimensionality* (Aggarwal, 2015). Therefore, our proposed solution simplifies the learning process, based on a small subset of features, avoiding high-dimensional data issues in the long run. As a result, although the acquisition of dynamic features is generally more complex and time-consuming than the collection of static features, they have proved to be more reliable, efficient, robust, and consistent over time, thus enabling the generation of a more effective detection system.

Finally, our characterization results demonstrate that the relevant system calls to discriminate benign applications do not show rapid variations in consecutive periods, whereas dominant feature sets for malware samples suffer radical changes. These results may help malware analysts get a general idea about the evolution of benign and malware samples and understand the reason behind concept drifts, thus improving the trust of the experts in the learning models. However, despite the advantages shown by system calls to generate an effective detection model, an expert may not derive a clear understanding of what type of app behavior is induced by each feature as an individual system call can be associated with different system functions. Static features such as permissions or API calls can benefit more from our characterization approach due to a more comprehensible mapping between these features and the behavior of the application. We consider the application of our methods to those features as one of the main directions in our future work.

6. Conclusions

The evolving nature of Android malware has been neglected by the majority of the machine learning-based detection methods proposed in the related literature, thus disregarding the degenerative impact of feature changes over time (i.e., concept drift). The reduced number of solutions that considered the impact of the *time* variable focused on the usage of API calls as input features. API calls can be used effectively to discriminate malware and provide a relatively good representation of its behavior. However, system calls, the most used dynamic features for Android malware detection, which allow capturing the real behavior of the apps at run-time, and are robust to obfuscation and encryption techniques, have not been considered in concept drift solutions.

This experimental study proposes a method that uses system calls data gathered on real Android devices to detect, characterize, and handle Android malware concept drift effectively.

Our proposed method minimizes model retraining and uses a pool of classifiers trained with recent data to adapt effectively to malware evolution. The experimental results evidence that system calls can effectively discriminate malware in the presence of concept drift using the proposed method, providing high-performance metrics for an extended period of time. More precisely, in a 7 year-long test, the proposed solution averaged 94.65% F1 score, 95.17% precision, 94.14% recall, and 89.49% specificity, proving the goodness of our solution to adapt and react to the concept drift issues that affect Android malware detection while keeping high-performance metrics. The proposed solution outperforms the state-of-the-art solutions for Android malware detection under concept drift conditions.

A critical issue to deal effectively with concept drift is the timestamp used to date the apps. In this study, distinct timestamps are analyzed and compared regarding concept drift-related performance. To the best of our knowledge, this is the first study on Android malware detection to perform such a comparison.

Lastly, the proposed solution allows the characterization of the changes in the data by analyzing the important features on the best classifiers. The observation of concept drift in different time horizons was used to describe the important features and determine their evolution and usefulness over time. In this regard, some features were found to have a prolonged (i.e., long-term) influence on the model performance, whereas others showed an impact limited to the short term (i.e., specific periods). This fact evidenced the existence of concept drift and provided insights into its character. More specifically, when the malware recognition task was analyzed (i.e., recall), it was observed that a small number of features had importance in each period, showing significant concept drifts and rapid feature importance changes. These facts were not observed for the benign software detection task (i.e., specificity).

The usage of the proposed method in combination with other relevant data features for Android malware detection, such as security *permissions*, remains part of our future work.

CRediT authorship contribution statement

Alejandro Guerra-Manzanares: Conceptualization, Methodology, Software, Validation, Investigation, Visualization, Writing – original draft, Writing – review & editing. **Marcin Luckner:** Conceptualization, Methodology, Investigation, Visualization, Writing – original draft, Writing – review & editing. **Hayretdin Bahsi:** Conceptualization, Writing – review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Aggarwal, C. C. (2015). *Data mining: The textbook*. Springer.

Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., et al. (2020). When malware is packin'heat; limits of machine learning classifiers based on static analysis features. *In Network and Distributed Systems Security (NDSS) Symposium 2020*.

Allix, K., Bissyandé, T. F., Klein, J., & Le Traon, Y. (2015). Are your training datasets yet relevant? In *International Symposium on Engineering Secure Software and Systems* (pp. 51–67). Springer..

Altmann, A., Tolo ̧si, L., Sander, O., & Lengauer, T. (2010). Permutation importance: A corrected feature importance measure. *Bioinformatics, 26*, 1340–1347. https://doi.org/10.1093/bioinformatics/btq134

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., & Rieck, K. (2020). Dos and don'ts of machine learning in computer security. *arXiv preprint arXiv:2010.09470*, .

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., & Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. *In Ndss, 14*, 23–26.

Barbero, F., Pendlebury, F., Pierazzi, F., & Cavallaro, L. (2020). Transcending transcend: Revisiting malware classification with conformal evaluation. *arXiv preprint arXiv: 2010.03856*, .

Breiman, L. (2001). Random Forests. *Machine Learning, 45*, 5–32. 10.1023/ A: 1010933404324.

Broersma, M. (2020). Android hit by 'incredibly sophisticated' malware. https://www.silicon.co.uk/workspace/android-sophiticatedmalware-344222.

Cai, H. (2020). Assessing and improving malware detection sustainability through app evolution studies. *ACM Transactions on Software Engineering and Methodology (TOSEM), 29*, 1–28.

Cai, H., Meng, N., Ryder, B., & Yao, D. (2018). Droidcat: Effective android malware detection and categorization via app-level profiling. *IEEE Transactions on Information Forensics and Security, 14*, 1455–1470.

Chebyshev, V. (2021). Mobile malware evolution 2020. https:// securelist.com/mobile-malware-evolution-2020/101029/.

Cruz, R. M., Sabourin, R., Cavalcanti, G. D., & Ren, T. I. (2015). Metades: A dynamic ensemble selection framework using meta-learning. *Pattern recognition, 48*, 1925–1935.

Elwell, R., & Polikar, R. (2011). Incremental learning of concept drift in nonstationary environments. *IEEE Transactions on Neural Networks*, *22*, 1517–.

Fu, X., & Cai, H. (2019). In *On the deterioration of learning-based malware detectors for android* (pp. 272–273). IEEE.

Guerra-Manzanares, A., Bahsi, H., & Nõmm, S. (2021). KronoDroid: Time-based hybrid-featured dataset for effective android malware detection and characterization. *Computers & Security, 110*, 102399.

Gözüaçık, Ö., & Can, F. (2020). Concept learning using one-class classifiers for implicit drift detection in evolving data streams. *Artificial Intelligence Review, URL:*. https://doi.org/10.1007/s10462-020-09939-x. 10.1007/s10462-020-09939-x

Guerra-Manzanares, A., Bahsi, H., & Nõmm, S. (2019a). Differences in Android Behavior Between Real Device and Emulator: A Malware Detection Perspective. In *Proceedings of the Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)* (pp. 399–404). IEEE.

Guerra-Manzanares, A., Nõmm, S., & Bahsi, H. (2019b). Time-frame Analysis of System Calls Behavior in Machine Learning-Based Mobile Malware Detection. In *2019 International Conference on Cyber Security for Emerging Technologies (CSET)* (pp. 1–8). IEEE.

Guerra-Manzanares, A., Nõmm, S., & Bahsi, H. (2019c). In-depth feature selection and ranking for automated detection of mobile malware. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP* (pp. 274–283). INSTICC, SciTePress.

Hu, D., Ma, Z., Zhang, X., Li, P., Ye, D., & Ling, B. (2017). The concept drift problem in android malware detection and its solution. *Security and Communication Networks, 2017*. https://doi.org/10.1155/2017/4956386

Japkowicz, N., & Shah, M. (2011). *Evaluating Learning Algorithms: A Classification Perspective*. New York, NY, USA: Cambridge University Press.

Jordaney, R., Sharad, K., Dash, S. K., Wang, Z., Papini, D., Nouretdinov, I., & Cavallaro, L. (2017). Transcend: Detecting concept drift in malware classification models. In *26th {USENIX} Security Symposium ({USENIX}*.

Kaspersky (2020). Mobile security: Android vs ios - which one is safer? https://www.kaspersky.com/resource-center/threats/ android-vs-iphone-mobile-security.

Kaspersky (2021). Machine learning for malware detection. https: //media.kaspersky.com/en/enterprise-security/Kaspersky-LabWhitepaper-Machine-Learning.pdf.

Ko, A. H., Sabourin, R., & Britto, A. S., Jr (2008). From dynamic classifier selection to dynamic ensemble selection. *Pattern recognition, 41*, 1718–1731.

Lei, T., Qin, Z., Wang, Z., Li, Q., & Ye, D. (2019). Evedroid: Event-aware android malware detection against model degrading for iot devices. *IEEE Internet of Things Journal, 6*, 6668–6680. https://doi.org/10.1109/JIOT.2019.2909745

Liu, F. T., Ting, K. M., & Zhou, Z. H. (2012). Isolation-based anomaly detection. *ACM Transactions on Knowledge Discovery from Data, 6*, 1–44. https://doi.org/10.1145/2133360.2133363

Liu, K., Xu, S., Xu, G., Zhang, M., Sun, D., & Liu, H. (2020). A review of android malware detection approaches based on machine learning. *IEEE Access, 8*, 124579–124607.

Lu, N., Zhang, G., & Lu, J. (2014). Concept drift detection via competence models. *Artificial Intelligence, 209*, 11–28. URL: http://dx.1016/.

Luckner, M. (2019). Practical web spam lifelong machine learning system with automatic adjustment to current lifecycle phase. *Security and Communication Networks, 2019*. https://doi.org/10.1155/2019/6587020

Maimon, O., & Rokach, L. (Eds.). (2005). *Data Mining and Knowledge Discovery Handbook. A Complete Guide for Practitioners and Researchers*. San Francisco, CA, USA: Springer.

Margara, A., & Rabl, T. (2018). Definition of data streams. In S. Sakr, & A. Zomaya (Eds.), *Encyclopedia of Big Data Technologies* (pp. 1–4). Cham.

Microsoft (2020). Sophisticated new android malware marks the latest evolution of mobile ransomware. https://www.microsoft.com/security/ blog/2020/10/08/ sophisticated-new-android-malware-marks-thelatest-evolution-of-mobile-ransomware.

Mohd Razali, N., & Bee Wah, Y. (2011). Power comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling tests. Journal of Statistical Modeling and Analytics, *2*, 21–33.

Mutz, D., Valeur, F., Vigna, G., & Kruegel, C. (2006). Anomalous system call detection. *ACM Transactions on Information and System Security, 9*, 61–93. https://doi.org/10.1145/1127345.1127348

Narayanan, A., Chandramohan, M., Chen, L., & Liu, Y. (2017). Context-aware, adaptive, and scalable android malware detection through online learning. *IEEE Transactions on Emerging Topics in Computational Intelligence, 1*, 157–175. https://doi.org/10.1109/TETCI.2017.2699220

Narayanan, A., Yang, L., Chen, L., & Jinliang, L. (2016). Adaptive and scalable android malware detection through online learning. In *In 2016 International Joint Conference on Neural Networks (IJCNN)* (pp. 2484–2491). https://doi.org/10.1109/IJCNN.2016.7727508

Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E. D., Ross, G., & Stringhini, G. (2019). Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security (TOPS), 22*, 1–34.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., & Cavallaro, L. (2019). {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In *28th {USENIX} Security Symposium ({USENIX} Security 19)* (pp. 729–746).

Rafter, D. (2021). Android vs. ios: Which is more secure? https://us.norton.com/internetsecurity-mobile-android-vs-ios-whichis-more-secure.html.

Reddy, R., Swamy, M. K., & Kumar, D. A. (2021). Feature and sample size selection for malware classification process. In *ICCCE 2020* (pp. 217–223). Springer.

Ramirez-Gallego, S., Krawczyk, B., Garcia, S., Wózniak, M., & Herrera, F. (2017). A survey on data preprocessing for data stream mining: Current status and future directions. *Neurocomputing, 239*, 39–57. https://doi.org/10.1016/j.neucom.2017.01.078

Ruiz-Heras, A., Garćıa-Teodoro, P., & Śanchez-Casado, L. (2017). ADroid: Anomaly-based detection of malicious events in Android platforms. *International Journal of Information Security, 16*, 371–384.

Seiffert, C., Khoshgoftaar, T. M., Van Hulse, J., & Napolitano, A. (2010). RUSBoost: A hybrid approach to alleviating class imbalance. *IEEE Transactions on Systems, Man, and Cybernetics Part A: Systems and Humans, 40*, 185–197. https://doi.org/10.1109/TSMCA.2009.2029559

Statista (2021a). Development of new android malware worldwide from june 2016 to march 2020. https://www.statista.com/statistics/680705/ global-android-malware-volume.

Statista (2021b). Distribution of leading android malware types in 2019. https://www.statista.com/statistics/681006/share-of-androidtypes-of-malware.

Statista (2021c). Mobile operating system market share worldwide february 2021. statcounter..

Suarez-Tangil, G., Dash, S. K., Ahmadi, M., Kinder, J., Giacinto, G., & Cavallaro, L. (2017). Droidsieve: Fast and accurate classification of obfuscated android malware. In *In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (pp. 309–320).

Townsend, K. (2020). How smartphones have become one of the largest attack surfaces. https://blog.avastcom/smartphones-and-increasingmobile-threats-avast.

Université du Luxembourg (2021). Androzoo - lists of apks. https://androzoo.uni.lu/lists.

Unuchek, R. (2018). Mobile malware evolution 2017. https://securelist.com/mobile-malware-review-2017/84139.

Xu, K., Li, Y., Deng, R., Chen, K., & Xu, J. (2019). In *Droidevolver: Self-evolving android malware detection system* (pp. 47–62). IEEE.

Yang, L., Guo, W., Hao, Q., Ciptadi, A., Ahmadzadeh, A., Xing, X., et al. (2021). CADE}: Detecting and explaining concept drift samples for security applications. *In 30th {USENIX Security Symposium ({USENIX} Security 21)*.

Zhang, X., Zhang, Y., Zhong, M., Ding, D., Cao, Y., Zhang, Y., et al. (2020). Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In *In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (pp. 757–770).

Zhao, L., Wang, J., Chen, Y., Wu, F., Liu, Y. et al. (2021). Droidmfc: A novel android malware family classification scheme based on static analysis. *arXiv preprint arXiv: 2101.03965*, .

Zhou, Y., & Jiang, X. (2012). In *Dissecting android malware: Characterization and evolution* (pp. 95–109). IEEE. https://doi.org/10.1109/SP.2012.16.

Zyblewski, P., Sabourin, R., & Wózniak, M. (2021). Preprocessed dynamic classifier ensemble selection for highly imbalanced drifted data streams. *Information Fusion, 66*, 138–154. URL: .org/.

Appendix 6

Publication VI
A. Guerra-Manzanares, M. Luckner, and H. Bahsi. Concept drift and cross-device behavior: Challenges and implications for effective android malware detection. *Computers & Security*, 120:102757, 2022

Computers & Security 120 (2022) 102757

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Concept drift and cross-device behavior: Challenges and implications for effective android malware detection

Alejandro Guerra-Manzanares[a,*], Marcin Luckner[b], Hayretdin Bahsi[a]

[a] Department of Software Science, Tallinn University of Technology, Estonia
[b] Faculty of Mathematics and Information Science, Warsaw University of Technology, Poland

ARTICLE INFO

Article history:
Received 25 October 2021
Revised 4 February 2022
Accepted 13 May 2022
Available online 19 May 2022

Keywords:
Concept drift
Android
Malware detection
Android emulator
Real device
Smartphone
Mobile security

ABSTRACT

The large body of Android malware research has demonstrated that machine learning methods can provide high performance for detecting Android malware. However, the vast majority of studies underestimate the evolving nature of the threat landscape, which requires the creation of a model life-cycle to ensure effective continuous detection in real-world settings over time. In this study, we modeled the concept drift issue of Android malware detection, encompassing the years between 2011 and 2018, using dynamic feature sets (i.e., system calls) derived from Android apps. The relevant studies in the literature have not focused on the timestamp selection approach and its critical impact on effective drift modeling. We evaluated and compared distinct timestamp alternatives. Our experimental results show that a widely used timestamp in the literature yields poor results over time and that enhanced concept drift handling is achieved when an app *internal* timestamp was used. Additionally, this study sheds light on the usage of distinct data sources and their impact on concept drift modeling. We identified that dynamic features obtained for individual apps from different data sources (i.e., emulator and real device) show significant differences that can distort the modeling results. Therefore, the data sources should be considered and their fusion preferably avoided while creating the training and testing data sets. Our analysis is supported using a global interpretation method to comprehend and characterize the evolution of Android apps throughout the years from a data source-related perspective.

1. Introduction

Threats originating from mobile malware create significant security incidents (Palmer, 2018; Yaswant, 2021) as mobile devices store an increasing amount of valuable data about individuals and enterprises. Android is the dominant operating system (OS) in the mobile OS market with a market share of 72% as of September 2021 (Statista, 2021). Due to its open nature and high prevalence, Android devices are constantly targeted by cybercriminals. For instance, according to Kaspersky, 98% of mobile banking attacks have been launched against these devices (Kaspersky, 2020). Malware spread remains a significant problem in Android OS despite the implementation of countermeasures by Google (2021) and Android original equipment manufacturers (OEMs) (Samsung, 2021). The detection of mobile malware is a challenging task with the traditional signature-based techniques used by most antivirus software due to rapid changes in the threat landscape and the emergence of new malware types. Machine learning approaches are seen as compelling solutions to address this bottleneck, especially when *zero-day* malware is considered (Fedler et al., 2013; Whitwam, 2021).

There exists a large body of research regarding the application of machine learning to mobile malware detection (Sharma and Rattan, 2021). However, the vast majority of the studies aim to prove the superiority of the proposed machine learning-based solutions on *static* data sets, neglecting the *dynamism* of the phenomenon and the obstacles that are inevitable to face in real settings (i.e., malware evolution and new trends). Such experiments with static data sets can give initial insights about the feasibility of machine learning algorithms to solve the problem but not deeper application-oriented knowledge.

In an organizational setting, machine learning models are incorporated into continuous processes that require a sustainable data analytics ecosystem. In this regard, a well-functioning data pipeline should be established, model life-cycles should be carefully managed (i.e., creating, updating, or replacing the models) and interpretation of model results should be shared among experts to generate trust between them and the machine learning models. Results could also be used for re-designing the model life-cycles if needed. When the malware detection problem is reviewed from this application perspective, for instance, considering the typical

* Corresponding author.
E-mail address: alejandro.guerra@taltech.ee (A. Guerra-Manzanares).

https://doi.org/10.1016/j.cose.2022.102757
0167-4048/

setting of a malware scanner vendor, it is apparent that practitioners must solve various hindrances such as preventing the detrimental impact of variability in data sources on detection performance and adapting the models to the ever-evolving threat landscape. Such relevant aspects cannot be handled by just optimizing the learning models for a static data set.

A data analytics process addressing mobile malware detection should consider the data pipelines that are fed from heterogeneous resources such as real devices of customers, honeypots, threat intelligence feeds, and sandboxes. Dynamic features obtained from the same sample may vary according to the data collection environment (e.g., real device or emulated device), which may potentially cause a degenerative impact on the learning models when the distinct data sources are not considered in the training and testing steps of the machine learning workflow (Guerra-Manzanares et al., 2019a). In (Alzaylaee et al., 2017), emulator and real phone issues are addressed using dynamically collected API calls and intent filters as data features. A similar approach focused on detection performance is used in Guerra-Manzanares et al. (2019a,b). However, the data sets used in these studies are too small and do not consider the changes in data over time, a significant variable affecting the performance of malware detectors. No other studies in the literature have taken data source variation into consideration.

Malware behavior is prone to change over time due to the intrinsic evolving nature of the problem domain, revolving around the constant attack and defense battle between malicious actors and defenders. The possible transformation of legitimate samples should not be underestimated either. Therefore, concept drift handling should be integral to a solution aiming to provide continuous effective detection. A phenomenon that has been addressed by a limited number of proposed implementations (Cai et al., 2019; Xu et al., 2019; Zhang et al., 2020). Furthermore, the incorporation of a reliable *time* feature into the data set, which is the centerpiece of concept drift modeling, has not been analyzed nor discussed in the related literature yet. The fundamental and challenging issue regarding the definition of timestamp *reliability* within the context of mobile malware detection must be addressed so that the timestamp that best grasps the behavior of malware and benign software is utilized to model data *drift* effectively.

Acquiring knowledge from data is an iterative process between the creation of a machine learning model and its analysis. The process of malware detection necessitates the involvement of various experts such as malware analysts. Thus, characterization of malicious behavior via interpretability constructs can enhance human-machine interaction and create a bidirectional feedback loop between experts and learning models.

In this paper, we deeply investigate a dynamic feature set derived from Android apps (i.e., system calls) within a concept drift model. More specifically, we explored the impact of data sources by creating and comparing models induced from data sets collected on real devices and emulators. Due to the central importance of timestamps to concept drift modeling, in addition to data sources, we explored the effect of distinct timestamping options on detection performance. We also performed characterization of concept drift using a global interpretability method to shed more light on the changes in malware and benign samples over the years. For our experimental setup, the *KronoDroid* data set was used (Guerra-Manzanares et al., 2021), which provides timestamped data encompassing all years of Android history (i.e., 2008–2020). We applied a sequential workflow that starts with a data preprocessing stage and continues with two different procedures: *concept drift detection* and *concept drift modeling*. The former addresses the question of whether concept drift exists in the data and, if so, what type of drift occurs, utilizing one-class anomaly models based on the Isolation Forest algorithm (Gözüaçk and Can, 2020), whereas the latter addresses concept drift by dividing the whole study period into data chunks and induces an adaptive learning model that dynamically selects the best ensemble model from a pool of classifiers for each chunk (Guerra-Manzanares et al., 2022; Zyblewski et al., 2021). The last stage of the workflow applies the permutation feature importance technique (Breiman, 2001) to provide a chunk-based characterization of concept drift results.

It is worth emphasizing that the optimization of detection performance of the concept drift model is not the main aim of this research. Our focus is on the evaluation of the impact of distinct data sources and timestamps on model performance. In this regard, we created a working concept drift modeling solution complemented by a characterization step to comprehensively analyze the impact of data source variation and timestamp alternatives on the continuous detection of mobile malware throughout the years.

To the best of our knowledge, this study is the first work that explores the impact of timestamp alternatives on concept drift modeling in mobile malware detection, a critical issue to consider for *drifting* data. Moreover, the comparison of learning models induced from emulator and real device data sets has not been performed for system calls yet. The characterization of concept drift is another noteworthy contribution of our study as it may help security practitioners to better comprehend the behavioral changes of malware and legitimate apps leading to the observed concept drift.

This paper is structured as follows: Section 2 references the state-of-the-art in Android malware detection while Section 3 provides the methodological description of this study. The main results are detailed in Section 4. Section 5 outlines the discussion points and limitations of this research while Section 6 summarizes the study and future work.

2. Related work

Static and *dynamic* features extracted from Android *apps* are used to induce effective machine learning-based Android malware detection systems (Liu et al., 2020).

Static features are collected without running the app, generally from the source code or the *apk* bundle. Features such as security permissions, API calls, and intent filters lie inside this category. Static features are fast and easy to collect in an automated fashion. However, the detection systems built based on them are prone to be bypassed by *zero-day* and sophisticated malware, especially when obfuscation and encryption techniques are used.

The collection of dynamic features requires the app to be executed, allowing for the capture of the *real* behavior of the running app in a *live* environment. Features such as system calls and network flow data can be acquired using this approach. The acquisition of dynamic features is generally time-consuming and challenging but they tend to generate more robust and effective detection systems.

2.1. Real device vs. emulator

System calls are the most commonly used dynamic feature for Android malware detection (Liu et al., 2020). System calls are the mechanism used by running software to request a service from the kernel of the underlying OS. They allow collection of the behavior of the application by capturing the information flow between the distinct OS layers (Dimjašević et al., 2016). Due to their dynamic nature, the acquisition of system calls features requires the execution of the app in a live Android environment. *Real devices* and *emulators* are used as execution devices for such purpose. A *real device* is an actual physical phone running an Android OS version whereas an *emulator* is a software running on a computer that simulates almost all the capabilities of a real device (Android, 2021).

There is no clearly dominant execution platform in the recent related literature. While some researchers prefer the usage of real devices for their experimentation, either using single (Amin et al., 2016; Saracino et al., 2018; Xiao et al., 2019) or multiple real devices (Alzaylaee et al., 2020; Vidal et al., 2017; Wang and Li, 2021; Wei et al., 2022), others advocate for the exclusive usage of emulators to perform their operations, using either specialized sandboxes for dynamic analysis (Feng et al., 2018; Han et al., 2020) or general-purpose Android emulators (Casolare et al., 2021; Dimjašević et al., 2016; Guerra-Manzanares et al., 2019c; Jerbi et al., 2020; Lin et al., 2013; Surendran et al., 2020; Vinod et al., 2019; Zhang et al., 2021).

Table 1 outlines relevant and recent studies in the research domain. As can be observed, distinct Android platforms (i.e., device types), combined with different data sources, dynamic features and algorithms, have been used with significant success for malware discrimination purposes. In this regard, the single usage of any of the approaches shows advantages and limitations.

Emulators are easy to deploy, manage, and they fit perfectly in automated analysis and detection systems (Dimjašević et al., 2016), enabling the mimicry of almost all real device capabilities in a wide variety of *virtual* devices and Android versions without actually having each real device (Android, 2021). However, malware with anti-sandbox evasion techniques can deceive emulators (i.e., the malicious behavior would not be triggered if a sandbox environment is detected) (Lindorfer et al., 2015). Although some solutions provide enhancements on this issue (Naval et al., 2015; Vinod et al., 2019), they generally provide limited interaction (i.e., specific triggering events might not be possible such as SMS messages or SIM card detection (Feng et al., 2018)) and fail to install apps that do not support x86 or x86-64 architecture libraries.

Real devices are more difficult to manage and integrate into automated systems. For instance, restarting to run every sample in a *clean* device can be time-consuming, *rooting* can *brick* the device, and ensuring the exact same conditions for all tests might not be possible (Lin et al., 2013). However, they provide full interaction with the app, they are inherently immune to anti-sandbox techniques, and they show much fewer incompatibility issues (Guerra-Manzanares et al., 2021).

In any case, the main underlying *axiom* in these studies is that the behavior of applications is fully consistent across devices (Lin et al., 2013) and Android versions (Burguera et al., 2011; Vidal et al., 2017) and, consequently, that the nature of the devices (i.e., emulators or real devices) and OS versions used do not really matter. This axiomatic assumption explains the absence of homogeneity on the selection criteria and the wide variety of devices/versions and approaches used in research setups. However, the studies that have experimented with both devices (Alzaylaee et al., 2017; Guerra-Manzanares et al., 2019a; 2019b) challenge the *validity* of this *cross-device behavioral consistency* postulate. For instance, in Alzaylaee et al. (2017), when *API calls* and *intents*, usually analyzed as static features, were captured dynamically, real devices were found to provide more reliable and stable features for malware detection than emulators, thus leading to a more effective detection outcome. However, when system calls are used as features, as in Guerra-Manzanares et al. (2019b) and Guerra-Manzanares et al. (2019a), the results show that emulators may provide better detection outcomes than real Android devices.

As can be observed, both kinds of devices have been widely used for Android malware detection purposes. The selection criteria are mainly based on the available resources and required flexibility under the assumption that app behavior is consistent across devices. Emulators are usually preferred to perform such operations due to their comparatively lower analysis cost, flexibility, and easier integration in automated analysis. A small number of studies have considered both kinds of devices in their experimentation, and their outcomes challenge the validity of the *consistent behavior*

Table 1
Relevant Android malware detection studies using dynamic features.

Reference	Device type	Data set	Data set size	Features	Algorithm/s used	Accuracy	Task
Xiao et al. (2019)	Real	Drebin + Google Play	B: 3536 + M: 3567	System calls	LSTM	0.94	Binary
Amin et al. (2016)	Real	MalGenome + Google Play	B: 227 + M: 1260	System calls	Experimental *goodness* threshold	0.87	Binary
Saracino et al. (2018)	Real	4 data sources	B: 9804 + M: 2800	Behavioral	Features/Misbehaviors correlation	0.96	Binary
Alzaylaee et al. (2020)	Real	McAfee Labs	B: 19,620 + M: 11,505	API calls + Intents	D-NN	0.95	Binary
Vidal et al. (2017)	Real	MalGenome/Drebin	B: 570 + M: 5130	System calls	Statistical hypothesis testing	0.96	Binary
Wei et al. (2022)	Real	MalGenome + Xi'an University	B + M: 2000	System calls	Weighted features threshold	0.96	Binary
Wang and Li (2021)	Real	–	B: 1275 + M: 1275	Kernel-related	NB, DT, ANN, k-NN	0.98	Binary
Feng et al. (2018)	Emulator	Drebin/Androzoo + Google Play	B: 13,806 + M: 10,213	Behavioral	Ensemble + Meta-classifier	0.97	Binary
Han et al. (2020)	Emulator	–	B: 3535 + M: 25,134	Behavioral	Linear combination of classifiers	N/S	Binary
Zhang et al. (2021)	Emulator	PlayDrone + Drebin	B: 2707 + M: 2978	System calls	CNN-Bi-LSTM-Attention	0.97	Binary
Lin et al. (2013)	Emulator	–	B: 400 + M: 102	System calls	Bayes theorem	0.96	Multi
Vinod et al. (2019)	Emulator	7 data sources	B: 3130 + M: 11,514	System calls	RF, Rotation Forest, AdaBoost	0.99	Binary
Casolare et al. (2021)	Emulator	Drebin + Google Play	B: 3462 + M: 3355	System calls	RF, SVM, MLP, CNN	0.89	Binary
Guerra-Manzanares et al. (2019c)	Emulator	Drebin/VirusTotal + APKMirror	B: 1,000 + M: 2,000	System calls	kNN, LR, DT, SVM	0.97	Binary
Surendran et al. (2020)	Emulator	Drebin/AMD/Contagio + Google Play	B: 1,250 + M: 1,250	System calls	NB, SVM, DT, RF, ANN	0.99	Binary
Jerbi et al. (2020)	Emulator	AMD/DROIDCat + Google Play	B: 1000 + M: 2000	API calls	Genetic Algorithm	0.99	Binary

assumption, opening the door for further exploration of the phenomenon. This research gap is explored thoroughly in this research by analyzing the same set of applications on distinct Android platforms and assessing the cross-device detection performance of induced models.

2.2. Concept drift analysis

The vast majority of prior related studies built and tested their proposed solutions for Android malware detection using static *snaphots* of data from Android history, usually using the same data sets. In this regard, *MalGenome* (Zhou and Jiang, 2012) and *Drebin* (Arp et al., 2014) are the most used data sets for Android malware research. Despite their relatively small size and being composed of *outdated* data (i.e., their most recent samples date back to 2012), they are still used as the main sources of malware in recent publications (Sasidharan and Thomas, 2021). Even though some studies (Cai et al., 2021; Gao et al., 2021) complement their data with more recent and larger data sets, such as the *Android Malware Dataset* (AMD) (Wei et al., 2017), to mitigate data-related issues (i.e., Drebin duplication (Irolla and Dey, 2018)) and increase the representativeness of the data set, they still rely on incomplete, relatively *old* (i.e., AMD's most recent sample is from 2016 and it provides samples for just 71 malware families), and *short* snapshots of malware data from the whole Android historical timeline (i.e., from 2008 to 2021). Furthermore, when using data sets for machine learning purposes, the common practice is to mix all the data and then split it randomly into two disjoint sets (i.e., train/test sets), thus disregarding apps' location in the historical timeline. This fact undermines the *historical coherence* and yields *significantly biased* and *historically incoherent* results (Allix et al., 2015; Pendlebury et al., 2019).

As a result, these issues pose serious doubts about the *generalization* capabilities and effectiveness of these solutions to detect evolved and recent malware.

Only a limited number of the related studies considered the usage of distinct and *historically coherent* snapshots of Android history for the train/test split. However, as they show significant time gaps between them (Guerra-Manzanares et al., 2019a; 2019b; 2019c), concept drift and its degenerative impact are neglected. Consequently, the *time* variable and malware evolution over time have been purposely ignored in the vast majority of current Android malware research studies.

As provided in Table 2, only a few studies dealing with Android malware detection have considered the concept drift issue and proposed machine-learning solutions that *adapt* to changes in the data, and are able to minimize its detrimental effect over time. Even though some general approaches have been proposed to detect data *drift* (Barbero et al., 2020; Jordaney et al., 2017; Pendlebury et al., 2019), all the proposed solutions dealt with API calls (Cai, 2020; Cai et al., 2019; Lei et al., 2019; Narayanan et al., 2016; Onwuzurike et al., 2019; Xu et al., 2019; Zhang et al., 2020), an inherently static feature but one that can also be acquired dynamically. None of the studies have dealt with system calls, a *pure* dynamic feature that enables us to capture the real run-time behavior of the app and which is robust to obfuscation and encryption techniques that can bypass static API-based detection systems.

2.3. Timestamps: when time matters

The central elements behind concept drift analysis are *timestamps*. Timestamps enable the temporal placement of the sample, which aims to provide a *reliable* temporal context. However, due to the lagging nature of the malware discovery process, this is not always possible or generates reliability issues. Even though some concept drift-related studies did not provide information about

Table 2
Concept drift-related Android malware detection research.

Reference	Time-frame	Data set	Data set size	Features	Algorithm/s used	Timestamp	Performance
Narayanan et al. (2016)	2014	7 data sources	B: 44,347 + M: 42,910	Graph-kernel	Online classifier	Compilation date	Acc: 0.84
Onwuzurike et al. (2019)	2010–2016	Drebin/Virushare	B: 8,447 + M: 35,493	API calls	RF, k-NN, SVM	First seen	F1: 0.99-0.87
Cai et al. (2019)	2009–2017	5 data sources	B: 17,365 + M: 16,978	API calls	RF	First seen	F1: 0.97
Jordaney et al. (2017)	2010–2014	Drebin + MARVIN	B: 133,127 + M: 14,739	Static	Conformal Evaluation	-	F1: 0.82
Xu et al. (2019)	2011–2016	AndroZoo	B: 33,294 + M: 34,722	API calls	5 online classifiers	Compilation date	F1:0.95-0.85
Lei et al. (2019)	2012–2018	PlayDrone/Google Play + VirusShare	B: 14,956 + M: 28,848	API calls	ANN	First seen	F1: 0.99-0.84
Pendlebury et al. (2019)	2014–2016	AndroZoo	B: 116,993 + M: 12,735	Static	Evaluation framework	Compilation date	F1: 0.91-0.82
Barbero et al. (2020)	2014–2018	AndroZoo	B: 232,848 + M: 26,387	Static	Conformal Evaluator	Compilation date	F1: 0.90-0.70
Zhang et al. (2020)	2012–2018	5 data sources	B: 290,505 + M: 32,089	API calls	Evaluation framework	First seen	F1: 0.92-0.68
Cai (2020)	2010–2017	VirusShare/AndroZoo + Google Play	B: 13,627 + M: 12,755	API calls	RF	Compilation date	F1: 0.92-0.72

the timestamp approach they used (Onwuzurike et al., 2019), in the ones that reported these data, some common timestamp approaches, although differently named, can be observed.

The *compilation date* is an *internal* timestamp that relates to the creation or compilation time of the *apk* bundle. Despite being appointed as the most reliable timestamp in the past (Pendlebury et al., 2019) and used in related research (Barbero et al., 2020; Cai, 2020; Pendlebury et al., 2019; Xu et al., 2019), it has become an *unusable* approach as most of the apps released nowadays have it set at 1980 (Luxembourg, 2021). Another internal timestamp proposed lately is the *last modification* timestamp, which refers to the most recent modification timestamp found in any of the *apk* inner files (Guerra-Manzanares et al., 2021). This feature was introduced in Guerra-Manzanares et al. (2021) which discusses the feasibility of four distinct timestamp approaches for Android malware detection.

Even though internal timestamps could be deemed as *accurate* approaches, they are prone to third-party manipulation which could lead to temporal misplacement. In this regard, more robust temporal approaches can be achieved using *external* timestamps. *Virustotal*'s *first seen*, also referred as *appearance* or *submission* time in the literature, dates the application with the *datetime* it was first received by the VirusTotal scanning service. This timestamp has been used in relevant Android concept drift-related studies (Cai et al., 2019; Lei et al., 2019; Zhang et al., 2020) as being based on external and reliable services, making it easy to acquire and more robust to alterations. However, it is prone to significant delay and time misplacements due to the required proactive behavior from the user to *timestamp* the app (i.e., submission of the file).

As can be observed, the timestamp approach emerges as a critical issue to properly handle data *drift* for effective Android malware detection. Despite that, it has been neglected by all the concept drift-related studies in the problem domain. In this research, we address this research gap by considering and evaluating distinct timestamps.

2.4. Explainability in android malware detection

Our work aims to understand and evaluate the decision process behind the concept drift model utilized for Android malware detection. *Explainability* or *interpretability* methods have been used to understand the decision processes used by the machine learning-based detection systems on their predictions (i.e., XAI). In this regard, Scalas et al. (2019) stated that research regarding ransomware detection should focus attention on the explainability of the predictions to review the model outputs for the purpose of better detection. They used explainability methods to find the most discriminant features analyzing packages, classes, and methods used in Android applications. Kinkead et al. (2021) pointed out the lack of research regarding explainability behind the predictions made by Android malware detection systems. In their study, the LIME algorithm was used to find the most important features for the classification task, and LIME activations were analyzed for specific malware families.

Karn et al. (2021) explored the usage of explainability methods on models based on system calls to classify anomalous cloud containers. The authors compared XAI techniques and concluded that not all have practical applications for malware detection (i.e., SHAP and LIME are efficient but, the LSTM autoencoder is less amenable for automated explanation extraction because of convergence instability). Iadarola et al. (2021) proposed a novel method based on image representations of Android apps used as an input for an explainable deep learning model designed for Android malware detection and malware family recognition tasks. In this work, we applied a post-hoc explainability method to characterize and analyze the evolution of mobile malware over time.

2.5. Contribution to the field

As a result, even though the related literature does not consider any cross-device behavioral differences, several studies found that the dynamic behavior of an app might not be fully consistent across Android platforms. This fact may lead to a degenerative impact on the models when data from distinct sources are mixed or not properly used. Furthermore, the *time* variable is usually neglected in Android malware detection studies, which poses a severe concern regarding the generalization capabilities of the proposed solutions, trained on outdated data, against new malware. Finally, the timestamp approach, a critical variable for effective concept drift handling, has not been properly considered in the concept drift-related studies.

The main contribution of this study is to shed light on those significant research gaps by assessing the cross-device behavioral differences using system calls for Android malware detection under the consideration of the time variable (i.e., concept drift), the analysis of distinct timestamp approaches, and the assessment of their impact on the machine-learning-based models over an extended period of time.

3. Methodology

3.1. Data set

The data set used in this research is *KronoDroid* (Guerra-Manzanares et al., 2021), a hybrid-featured, timestamped, and labeled Android data set that includes malware and benign samples for all years of Android history (i.e., 2008–2020). Kronodroid is split into two data sets with different sizes, according to the acquisition device used for the dynamic features it provides (i.e., system calls). Therefore, emulator and real device-related data sets compose the full *KronoDroid* data set. This device-related split makes it an ideal data set to perform a behavioral comparison between emulators and real devices. However, as the sizes of the data sets are different, as not all apps were run in both devices, to perform a sound comparison of the dynamic profiles, just the intersection between the two data sets was selected using the *hash* attribute. As a result, the intersection data set, used in this research, was composed of 28,343 malware samples and 34,981 benign apps.

The KronoDroid data set provides four possible timestamps per record: *last modification*, *earliest modification*, *first seen VT*, and *first seen in the wild*. Based on their reliability and prevalence among the data points, two timestamps were selected for this study: *last modification* and *first seen VT*. The *earliest modification* and *first seen in the wild* timestamps were discarded due to the inaccurate nature of the former (i.e., many apps had a 1980 value) and the high ratio of missing data of the latter (i.e., not available for most of the apps) (Guerra-Manzanares et al., 2021). The *last modification* timestamp locates the app within the Android history timeline according to the most recent modification timestamp retrieved among the app *inner* files. In contrast, the *first seen VT* reports about the date and time when the app was submitted to *VirusTotal* for the first time.

The analysis of concept drift-related issues requires the usage of the *historical* context based on app timestamps. In this regard, there is no unambiguous approach to determine an app's temporal location with complete reliability and accuracy. Due to its generation mechanism, the *first seen* timestamp is prone to important delays as it depends on submission by users to VirusTotal. This timestamp heavily depends on the usability and popularity of the service to get timely notification of malware samples based on the users' proactive behavior. Therefore, it can be hypothesized that the reliability of the *last modification* timestamp, when it has not been tampered with, should be greater than the *first seen* timestamp in terms of accurately positioning the app in the historical

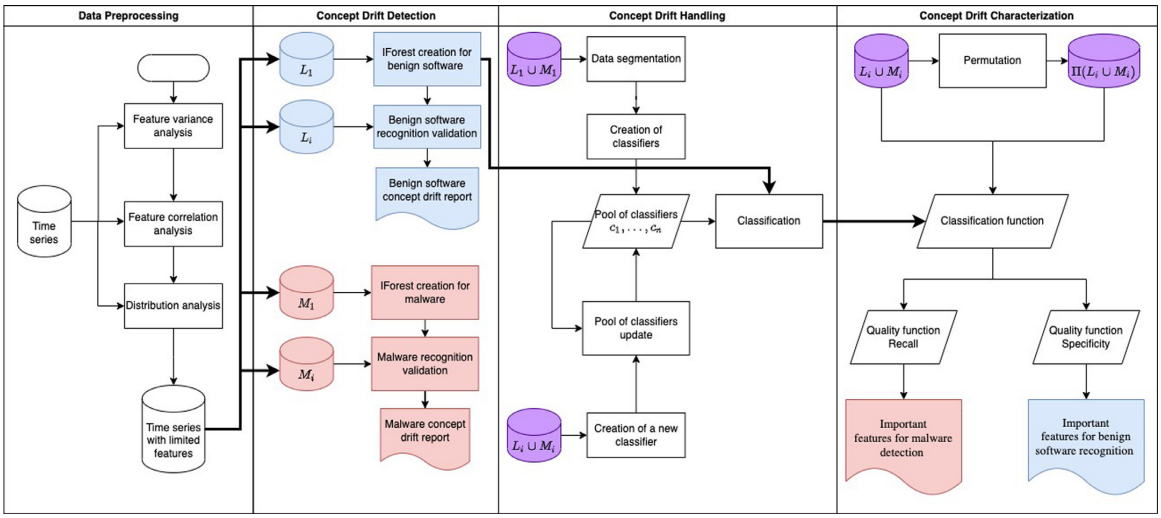


Fig. 1. Depiction of the methodological workflow followed in this research.

timeline. Despite that, in this study, we used and experimentally compared the reliability of these two approaches to deal with concept drift.

Lastly, the behavioral features collected per app in both data sets are system calls, also named as *kernel calls* or *syscalls* for short. The whole feature set is composed of 288 system calls. As Guerra-Manzanares et al. (2021) emphasizes, some of the system calls are device-related; thus the feature sets may not be consistent across Android platforms. For instance, the emulator feature set is composed of 212 system calls, whereas the real device feature set includes these features and more system calls, being extended to 288 features. Therefore, to perform a sound comparison, this comparative analysis uses both feature sets in the experimental approach, referenced distinctively as *emulator* or *reduced* feature set (i.e., 212 system calls) and *real* or *extended* feature set (i.e., 288 system calls). It is worth mentioning that when the extended feature set is used to characterize the emulator data, the values for the syscalls that belong exclusively to the real device set and which are not present in the emulator data (i.e., 76 features) are filled with zeroes. As provided by the Kronodroid data set, for each sample, the value for each feature provides the number of times the specific system call feature was used by the specific application at run-time. Thus, the input vector for the induced classifiers was composed of numeric values reflecting the absolute frequency of each system call invoked per application.

3.2. Workflow

The methodology used in this study is composed of 4 sequential phases. They are summarized in Fig. 1 and briefly explained as follows:

1) *Data Preprocessing*: zero-valued and redundant features were removed from the initial feature sets. The distributions of the remaining features were assessed using normality tests.
2) *Concept Drift Detection*: anomaly detection models were induced to assess the existence of concept drift in the data.
3) *Concept Drift Handling*: an existing solution for data streams (Zyblewski et al., 2021), was slightly customized as described in Guerra-Manzanares et al. (2022) and used to address the concept drift issue in Android malware data. Different combinations of training and testing data sets belonging to distinct data sources were evaluated with different timestamps.
4) *Concept Drift Characterization*: the analysis of changes in feature importance over time was used to characterize the observed concept drift.

A more thorough explanation of the stages is provided in the following paragraphs.

3.2.1. Data preprocessing

Machine learning heavily relies on data quality to build effective models. The removal of redundant and irrelevant features is an essential step to improve data quality within the machine learning workflow. This step aims to remove noisy, repeated, and unimportant features within the data set that may harm the classifier performance during the model's training. Three sequential steps were performed in this phase:

1) *Variance analysis*: sample variance was calculated for all features. A zero variance value, reporting no variability, was obtained for features that had constant or zero values for all samples and labels. Therefore, zero variance features were removed as they did not provide any relevant information to describe the data.
2) *Correlation analysis*: *Pearson's correlation coefficient* (r) was calculated pairwise for all features. Highly correlated features (i.e., $|r| \geq 0.80$) were dropped. This step aims to remove redundant data, a critical step for model building and the characterization methods used in this study. Correlated features may disturb the outcomes of perturbation-based global interpretability methods; thus, their removal is essential to have more reliable characterization results (Molnar et al., 2020).
3) *Distribution analysis*: the adherence of each feature distribution to the *Gaussian* distribution was assessed using statistical tests. The adherence of the feature distribution to normality is useful to assess the techniques used in posterior steps.

Once the data preprocessing step was concluded, the resulting feature sets (i.e., emulator and real device feature sets) were used in the following stages to tackle concept drift and analyze cross-device behavioral differences.

3.2.2. Concept drift detection

This phase aims to assess whether the use of concept drift handling is necessary, that is, if there is significant *drift* in the data.

In a continuous analytics process, each new observation can be represented by $c_i = (x_i, y_i)$, where $x_i = (x_i^1, x_i^2, \ldots, x_i^n) \in \mathbf{X}$ is the feature vector and $y_i \in \mathbf{Y}$ is the target label. The incoming observations are aggregated into chunks, probes of the same size, or collected during a similar period (i.e., six months time-frame). Let us assume that features from two chunks can be described by distributions F and F'. *Feature drift* is defined if the null hypothesis H_0 that F and F' are identical can be rejected (Lu et al., 2014), that is, they are *significantly different* distributions. Despite this clear statistics-based definition, feature drift can be hard to detect in real data using statistical methods. For instance, Mutz et al. (2006) and Ruiz-Heras et al. (2017) showed that Android system calls could not be modeled using Gaussian distribution. Moreover, feature drift detection is not very relevant from a practical point of view. A more relevant phenomenon is *concept drift*, which occurs when feature drift leads to a change in $\hat{y}$, the target estimation value provided by a predictive model.

Relying on the aforementioned definitions and reservations, concept drift can be detected experimentally. Let us take two series of data M_i and L_i ordered in n subsequent chunks. The series describe malware and benign software, respectively, with the value of i referring to the order in the sequence, $1 \leq i \leq n$.

Next, let us define the most important discriminators among the features using the following procedure. The data from the first chunk $M_1 \cup L_1$ was balanced using a random oversampling method (Seiffert et al., 2010) to avoid over-representation of any of the classes. Then, the classes were discriminated using the Random Forest (RF) algorithm (Breiman, 2001), a fast and reliable ML algorithm tested in similar scenarios showing outstanding performance (Guerra-Manzanares et al., 2019a; 2019b). The most relevant features of this initial classifier were selected using the permutation feature importance technique (Altmann et al., 2010). Only the features with positive mean importance were selected, thus generating the *important* feature set. In this regard, if the *important* set of features can obtain high performance on the initial set $L_1 \cup M_1$, a relevant question is whether the performance level can be kept for $L_i \cup M_i$ where $i > 1$.

To test this issue, *one-class* anomaly detection models trained separately on L_1 and M_1 were employed. The usage of one-class algorithms eliminates the class relations influence. The Isolation Forest algorithm proposed by Gözüaçk and Can (2020) was used as the detection algorithm. The detectors trained on initial-period data (i.e., $i = 1$) were tested on the subsequent L_i and M_i data sets described by the *important* feature set to calculate the ratio of observations recognized as part of the modeled class in the given chunk. The decrease in ratio signals the occurrence of concept drift, which occurs when the initially selected *important* features are not able to correctly model the analyzed phenomenon in the test data.

3.2.3. Concept drift handling

The concept drift problem is usually identified in data streams (Aggarwal, 2015; Margara and Rabl, 2018). However, Android malware detection shows related characteristics and faces similar issues; thus, a solution to handle emerging concept drift for data streams could be applied. In Zyblewski et al. (2021) an algorithm to address concept drift issues in data streams split into *data chunks* was proposed. The method uses a pool of classifiers trained on past data to make predictions about new data samples. During the prediction process, the best ensemble of classifiers is dynamically selected to perform accurate predictions. Furthermore, the pool is modified to introduce classifiers trained on new data and remove low-performance models, aiming to keep high performance over time by updating the pool of classifiers with new and evolved data. The pool update procedure enables the detection system to handle concept drift effectively.

To apply the original solution described in Zyblewski et al. (2021) for Android data analytics, the following changes were applied, as proposed in Guerra-Manzanares et al. (2022).

- The classifier pool was full and ready from the first data chunk. This fact avoids waiting for S chunks to gradually fill the classifier pool until its completeness (i.e., the S hyper-parameter refers to fixed pool size), as proposed in the original solution.
- The pool of binary classifiers is supported by an anomaly detection model to improve the recognition of benign software. This improvement was made on the basis of the experimental research that evidenced a more consistent profile over time in benign data than in malware data.

The proposed classification method, based on dynamic ensemble selection, is used in this research as a tool for concept drift handling and characterization.

3.2.4. Concept drift characterization

The main aim of this investigation is not the optimization of concept drift detection but to use the concept drift handling method to analyze changes and differences in Android malware detection when distinct data sources are used over time. For this purpose, the *permutation feature importance* technique (Breiman, 2001) was employed to analyze whether the *important* feature sets were significantly different among data chunks and for distinct data sources.

The permutation feature importance technique is an alternative method to the built-in Random Forest's importance estimation (Maimon and Rokach, 2005). The method is defined as follows. For a matrix of feature values $\mathbf{X}$ with rows $\mathbf{x}_i$ given each of N observations and corresponding response y_i, $\mathbf{x}_i^{\pi,j}$ is a vector achieved by randomly permuting the jth column of $\mathbf{X}$. The method determines the *importance* of a feature for the model by assessing the decrease in the model's performance after a random permutation for the specific feature is performed while keeping the other features unchanged. According to Altmann et al. (2010) and due to the stochastic nature of the technique, the permutation process should be repeated at least 50 times to achieve stable results. For a loss function L, the importance VI_j of the jth feature is defined as the difference between the loss calculated using pseudo-random values and the original data.

The concept drift characterization method uses the classification function f_t on data X_t from period P_t. Next, the analysis observations X are taken from the set $\mathbb{U}_{l=t+1}^{l+h} X_j$ where h declares a time horizon for the analysis (e.g., 3 months). The procedure is summarized in the following equation:

$$VI_j^{\pi}(t) = \frac{1}{N} \sum_{\substack{i=1, \\ x_i \in \mathbb{U}_{l=t+1}^{l+h} X_l}}^{N} Q(y_i, f_t(x_i)) - Q\left(y_i, f_t(\mathbf{x}_i^{\pi,j})\right), \tag{1}$$

where $Q(.) = 1 - L(.)$ is a quality function such as:

- *F1 score*, a comprehensive metric for malware detection performance on imbalanced data sets defined as:

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{2}$$

- *Specificity (True Negative Rate)*, which provides the benign software recognition performance (i.e., negative label) and it is calculated as:

$$TNR = \frac{TN}{TN + FP} \tag{3}$$

Table 3
Data preprocessing results.

Preprocessing stage	Results	
	Emulator	Real device
Initial Set	212 syscalls	288 syscalls
Variance Analysis	90 constant	160 constant
Correlation Analysis	28 high-correlated	31 high-correlated
Distribution Analysis	0 normal	0 normal
Final Set	94 syscalls	97 syscalls

- *Recall (True Positive Rate)*, a measure of the quality of malware detection (i.e., positive label) defined as:

$$TPR = \frac{TP}{TP + FN} \tag{4}$$

where *TP* (i.e., true positive) refers to the number of correctly recognized malware in the test set. *TN* (i.e., true negative) reflects the number of correctly recognized benign software in the test data. *FP* (i.e., false positive) provides the number of incorrectly recognized malwares in the test set, and *FN* (i.e., false negative) provides the number of incorrectly recognized benign data points in the test samples.

4. Results

4.1. Data preprocessing

The results obtained after the application of each preprocessing step are summarized in Table 3 and explained in the following paragraphs.

The initial feature sets, related to each device, were composed of 212 features for the emulator and 288 features for the real device. After *variance analysis*, 122 syscalls showed non-zero variance for the emulator case and 128 for the real device case. Thus, 90 features were removed from the emulator feature set and 160 from the real device feature set. The remaining features on each set were further processed and highly correlated features (i.e., $|r| \geq 0.80$) were removed. As a result, the final feature sets were composed of 94 features for the emulator and 97 for the real device.

The *normality* tests applied to the final sets of features showed that no feature was normally distributed. Figs. 2 and 3 show the distributions of features included on both final sets as illustrating examples (i.e., red for malware samples' values and green for benign samples).

As can be observed, both features (i.e., *getuid32* in the left graph and *ioctl* in the right) show a positively skewed distribution in both Android platforms, and, consequently, a non-normal distribution. Furthermore, there is a remarkable difference in the shape of the distributions for the same feature on each of the devices. Analogous differences were spotted for all syscalls distributions in both platforms. Therefore, these differences arise as initial support to challenge the assumption of cross-device consistent behavior.

4.2. Concept drift detection

The KronoDroid data set provides timestamped data for the whole Android history (i.e., 2008–2020). Initially, the data was split into 6-month data chunks for both timestamps. The first period with enough data to build a classifier for both timestamps corresponds to the second semester of 2011. Even though the KronoDroid data set provides data from previous years, the selected period was preferred in order to avoid biased results due to the small number of samples belonging to the previous periods in the data set. The scarcity of data samples for the 2008–2010 time frame is consistent with the actual threat landscape timeline, as the first Android malware was discovered in 2010 (Sophos, 2017). Therefore, the second semester of 2011 was selected as the *initial period* for both timestamps.

Table 4
Important feature sets ranges.

Timestamp	Data	Min	Max
Last Modification	Emulator	28	31
	Real Device	29	32
First Seen	Emulator	16	21
	Real Device	16	26

Table 5
Top-10 features ranking.

Emulator		Real device	
Last mod	First seen	Last mod	First seen
rt_sigprocmask	rt_sigprocmask	epoll_ctl	clock_gettime
fcntl64	getuid32	futex	SYS_329
futex	ioctl	SYS_329	writev
getuid32	recvfrom	clock_gettime	epoll_ctl
ioctl	read	writev	getuid32
write	futex	ioctl	write
read	write	write	close
writev	fcntl64	getuid32	gettimeofday
recvfrom	prctl	munmap	ioctl
pread64	fstatat64	read	connect

The initial period was composed of 8378 samples for the last modification timestamp (i.e., 6672 benign, 1706 malware) and 2595 instances for the first seen timestamp (i.e., 2133 benign, 462 malware). In both cases, the data was imbalanced towards the benign class. The data sets were balanced using a random oversampling technique, and the data was used to induce Random Forest classifiers with 300 estimators. Both classifiers provided an accuracy of over 0.95 on test data.

In order to select the most relevant features for each classifier, the permutation feature importance technique was applied to the training data (i.e., 500 permutations per feature). Only features with positive average importance were selected, as they reflect the actual impact on the model's performance. The results were averaged and ranked. Due to the stochastic nature of the permutation technique, a distinct amount of features might be part of the *important* feature sets on every trial. Therefore, ten trials were performed. Table 4 provides the ranges of the number of important features observed after the iterations.

As can be observed, a smaller set of important features was observed using the first seen timestamp for both devices' data. However, the range and *inner* variability in the selected sets were greater for the first seen timestamp than for the last modification timestamp. Besides, the last modification timestamp showed a much more consistent feature set selection across trials, thus showing greater stability on the number and the composition of the sets of important features. As a descriptive example, the top 10 features for each timestamp and device combination are provided in Table 5 in decreasing importance order. These results were obtained by averaging the importance ranking positions on each iteration. The two columns on the left in Table 5 provide the information about emulator data features sets for each timestamp. The two right-most columns show the same information for the real device data. For a better comparison, data related to the same timestamp are displayed in the same color (i.e., grey for last modification and white for first seen). Features observed in all feature sets are highlighted in blue.

As can be seen in Table 5, the feature sets differ not only between timestamps but more remarkably between Android platforms. More precisely, the usage of distinct timestamps in the same device produced relatively similar feature sets, mostly chang-

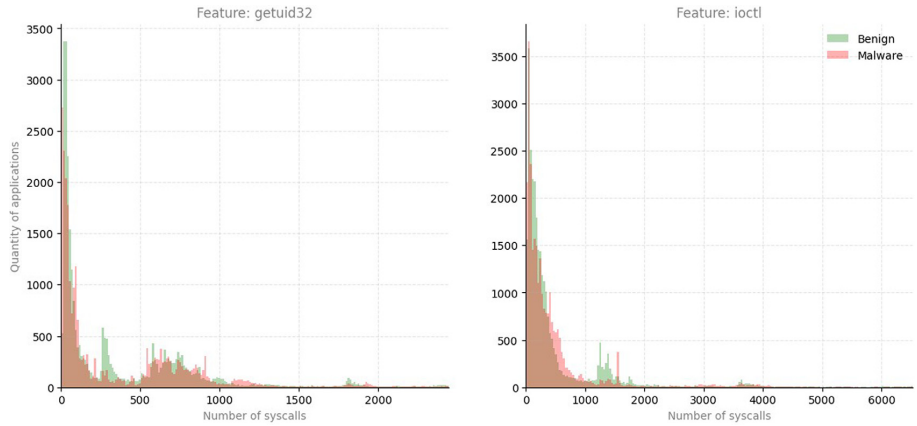


Fig. 2. Emulator features distributions.

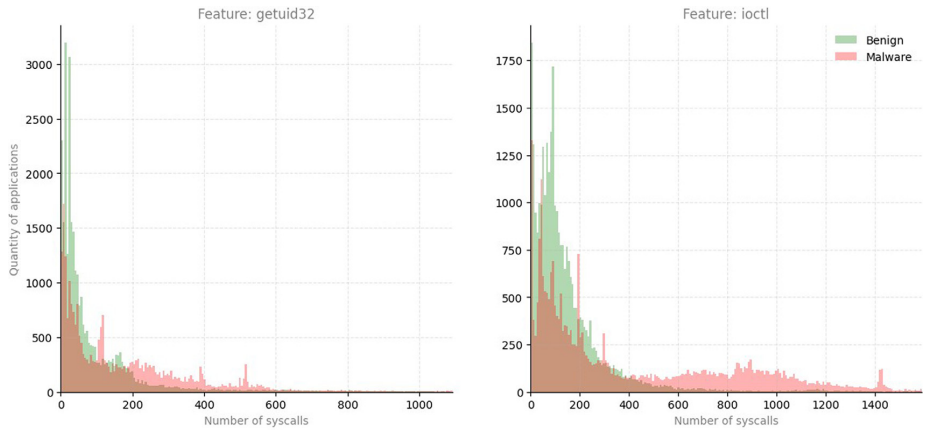


Fig. 3. Real device features distributions.

ing in order. But when the feature sets are compared between devices, the differences are significant. For instance, the most important feature in the emulator is *rt_sigprocmask* for both timestamps, whereas in the real device this feature is not found in the top 10 for any timestamp. Similarly, *clock_gettime* shows noticeable importance in the real device but not in the emulator. Three features are common in all rankings but located on distinct positions, thus showing different importance (i.e., discriminatory power). Furthermore, architecture-related syscalls appear to have notable importance in both cases, as it is evidenced by the high ranking of *SYS_329* for the real device (i.e., ARM architecture) and *fcnlt64* for the emulator (i.e., x86_64 architecture). Therefore, the results provided in Table 5 suggest that the timestamp selected might cause differences in the relevant feature set, mostly related to the order, but that more significantly, the data source can have a critical impact on the definition of the feature sets.

These initial differences are further explored by assessing concept drift in the data. In order to test the data changes, *one-class* anomaly detection models (i.e., one for malware detection and another for benign software detection) were built using the *minimal* important feature sets found using permutation feature importance as the feature selection technique. The *minimal* feature sets were constructed using the smallest *important* feature set among all iterations for each device and timestamp combination (i.e., the lower boundary (min) reported in Table 4). The rationale behind the anomaly detection test is explained as follows. If the phenomenon is stationary, meaning that the initial-period features, even with varying discriminatory power, could be consistently used to perform effective *class* discrimination in future data, the anomaly models built should show high performance over time. However, if the phenomenon evolves, meaning that important features for effective discrimination are prone to change, the anomaly model performance should drop or fluctuate significantly over time. Therefore, in these models, data *drift* is detected as an *anomaly* with respect to the initial data.

Anomaly detection models were induced using 5, 10, and all features of the minimal sets. The results of the initial-period anomaly models evaluated using 6-month data chunks from consecutive periods in the 2011–2020 time-frame are shown in Figs. 4 and 5 for emulator data and real device data respectively. For

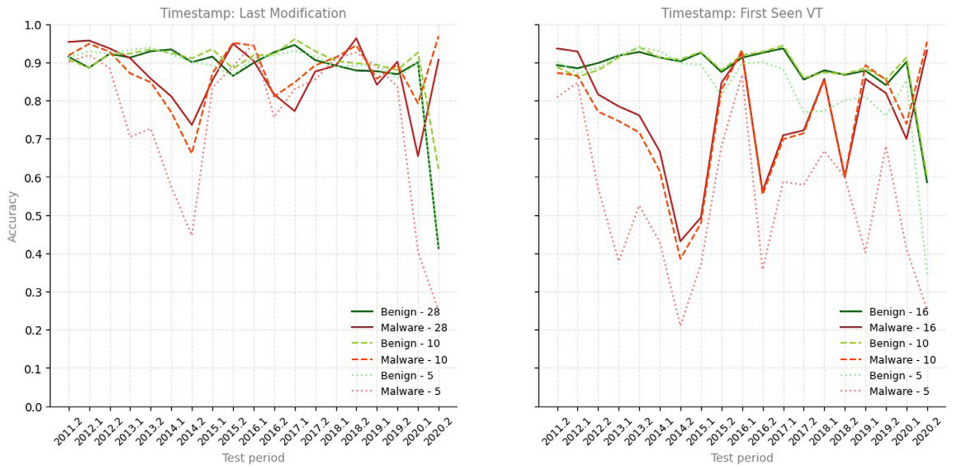


Fig. 4. Emulator anomaly detection models.

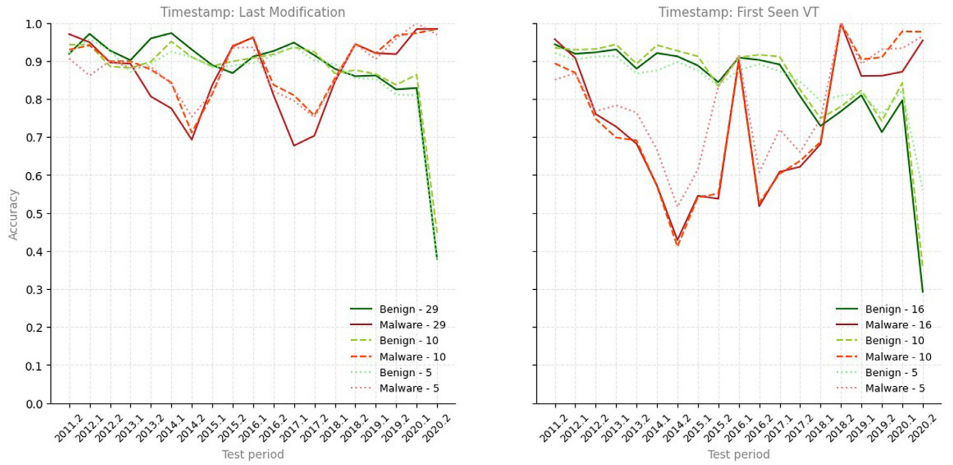


Fig. 5. Real device anomaly detection models.

each device-related graph, six lines are depicted, corresponding to anomaly models induced for each class using different feature sets (i.e., red for malware and green for benign software) under one timestamp. Model accuracy is provided in the vertical axis while the time-frame of the test data is provided in the horizontal axis. In this regard, the test periods correspond to 6-month consecutive chunks where the added suffix *.1* and *.2* to each year refer to the first six months and to the last six months respectively.

As can be observed in Fig. 4, the data obtained from the emulator does not show data *drift* for benign software. The accuracy fluctuates in a tight range for all test chunks, except for the last one. However, in the case of malware, concept drift is present in the form of *blips* (i.e., rapid decrease of accuracy deep under the reference level outlined by the benign software) (Ramírez-Gallego et al., 2017), for instance in the 2014.2 and 2020.1 periods. Furthermore, the blips are more pronounced for the first seen timestamp. The remarkable differences observed between both timestamps deserve an in-depth analysis, which is presented in Section 4.3.

In the case of emulator data, a reduced number of important features (i.e., 10) yielded similar results as the whole set of essential features. However, the reduction to five features decreased accuracy rapidly, especially for the observed *blips*. This situation is not observed in the real device data, where five features showed better performance than the whole minimal feature set.

For the real device data, the benign data shows a slightly decreasing tendency over time. Despite that, the performance is over 0.80 for most of the analyzed period. In the case of malware, data drift is observed in the form of repeated dips around 0.70 performance area for the last modification timestamp and reaching lower values for the first seen timestamp. Again, the performance dips are more pronounced for the first seen timestamp, analogously to the emulator case.

Substantial differences are observed between the emulator and real device data, depicted in Figs. 4 and 5 respectively. Firstly, the usage of distinct timestamps provides remarkably different performance for both cases. Overall, data blips for the real device malware data are deeper than for emulator data. However, emulator malware data show more blips. Despite that, in both cases, the last modification timestamp provides fewer and shallower dips than the first seen timestamp. Secondly, the reduction in the size of the feature sets increased accuracy for most of the chunks in the real device data but not in the emulator. Lastly, accuracy in the case of real device benign data shows a decreasing tendency, especially for the first seen timestamp. Such a tendency may obscure the analysis of concept drift in a classification task.

As a result and regardless of the data and timestamp used, the pronounced fluctuations in the accuracy performance for malware in all cases evidence that the initial set of important features did not maintain its discriminatory power over time. It lost it in some periods (i.e., blips, where some other features became important) to regain it in some others (i.e., peaks). This fact clearly manifests the existence of concept drift in the data. Furthermore, when using the first seen timestamp the observed *drifts* seem to be more significant (i.e., deeper dips) than when the last modification timestamp is used. This fact indicates the generation of a more pronounced drift in the data when the first seen timestamp is used, which might be caused by the delayed nature of this timestamp and the consequent data misplacement.

The observations extracted from Figs. 4 and 5 evidenced that concept drift emerges as a significant threat to the performance of detection models over time. The next stage in our workflow is to address the situation with a dedicated classifier and use it to perform a deeper analysis of the data.

4.3. Concept drift handling

The concept drift-handling detection system, based on the dynamic selection of the best ensemble of classifiers from a pool of classifiers and its constant update as explained in Section 3.2.3, was used to analyze malware and benign data.

In order to explore the phenomenon from all possible perspectives, the solution was applied using different combinations of training and testing sets that were described using both feature sets (i.e., reduced and extended feature sets). The usage of emulator and real device data allowed us to analyze differences between data sources and the usage of both feature sets enabled us to explore the effect of features in detection performance. Furthermore, both timestamps were also used for every feature set and data source combination. For instance, when emulator data was used as the *train* set with the last modification timestamp, 4 distinct combinations were tested by using the 2 possible feature sets as data descriptors and the 2 data sources as *test* set. These multi-testing scenario results are reported in Fig. 6. The analysis of the phenomenon taking all possible permutations of the variables into account (i.e., feature set, device, and timestamp) enriched the analysis of differences between data sources, the impact of feature sets, and the reliability of timestamps. More importantly, it enabled us to ensure unbiased results as no assumption was performed.

For the sake of deeper exploration of the phenomenon, a greater level of granularity was used to better capture emerging concept drift. The data was split into quarter-year data chunks limited to 4000 samples, thus data were analyzed for each quarter of the 2011–2018 time frame. F1 score performance metric was calculated for each period using the classification model, composed of a dynamic ensemble of n classifiers trained during previous periods (i.e., $n = 12$ yielded the best performance in our experimental setup).

The obtained results illustrate changes in the quality of malware detection among periods. The results for the last modification timestamp using emulator data as the train set are provided in Fig. 6. Fig. 7 shows the results obtained when real device data were used as the train set for the same timestamp. As can be observed, the 4 possible combinations of test data variables are plotted. Disregarding the train data and timestamp, test data are always defined by a data source (i.e, emulator or real device) and a data descriptor set (i.e., *reduced* or *extended* feature set used to describe the data). In the cases where the test data source differs from the train data source, it enables us to explore cross-device performance, whereas the usage of distinct feature sets provides information about the discriminatory capabilities of a larger feature set versus a reduced feature set. In the figures, the feature set is referred to as the original source of data it belongs to (i.e., they are architecture-related features). For instance, in Fig. 6, *Real* refers to the extended set of features and *Emu* to the reduced set of features. This denomination is preferred to properly relate the feature set impact to the data source. The same information is provided in Fig. 8 and Fig. 9 for the first seen timestamp. More precisely, Fig. 8 uses emulator data as the train set whereas Fig. 9 uses real device data to train the model. In these graphs, the horizontal axis reports the quarter analyzed, and the vertical axis the corresponding F1 performance of each test set on the trained model. It is worth noting that when using the first seen timestamp, a greater number of apps were dated in the period 2012–2013. The higher prevalence of malware for the first seen timestamp in this period might have been caused by the expansion and increased popularity of *VirusTotal* service during that time (VirusTotal, 2012). Due to the 4000 samples per chunk constraint, and to not miss emerging concept drift, more than one chunk was processed per quarter. Therefore, for this timestamp, in the 2012–2013 time frame the relation chunk-quarter was exceptionally bypassed to analyze all the available data. This is reflected by the greater separation in the horizontal axis for this specific time frame. These additional data chunks are provided for the sake of completeness and as evidence of the different temporal alignments for the same data sets when using different timestamps. For the sake of interpretability of the graphs, test data source and feature set are plotted distinctively. More specifically, the line color indicates the source of the test data (i.e, blue for emulator and yellow for real device) and the line style reflects the feature set used to describe the train/test sets (i.e., solid for the extended feature set and dashed for the reduced feature set).

When the last modification timestamp is considered, the performance of the proposed method to deal with concept drift is relatively stable. The method obtains over 0.80 F1 score in most of the studied timeframes especially when the model is tested with data from the same source as the training set as evidenced by the blue lines in Fig. 6 and the yellow lines in Fig. 7. The results obtained using different feature sets are similar. This fact indicates the goodness of the reduced feature set (i.e., emulator features) to provide similar performance as the extended feature set and that the zero-filling for missing data when the real device feature set is used to describe emulator data does not significantly harm the model performance. When the model is trained with real device data but tested with data from the emulator using real device features as descriptors (i.e., extended feature set), the results obtained are the worst for this timestamp. In this case, the feature set seems to notably impact the performance, as when real device data are used for training with emulator-based features (i.e., reduced feature set), better results are observed. This is confirmed by the fact that the performance *blip* observed in 2012-Q3 when the extended feature set is used it is not observed when the reduced feature set is used. More interestingly, the dip in 2012-Q3 is just observed

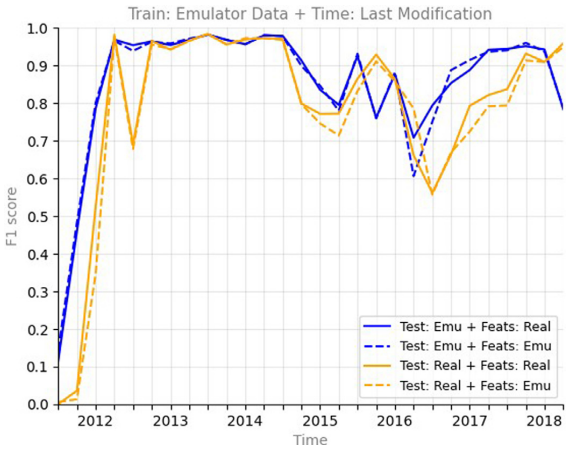


Fig. 6. Last Modification timestamp - Training Emulator.

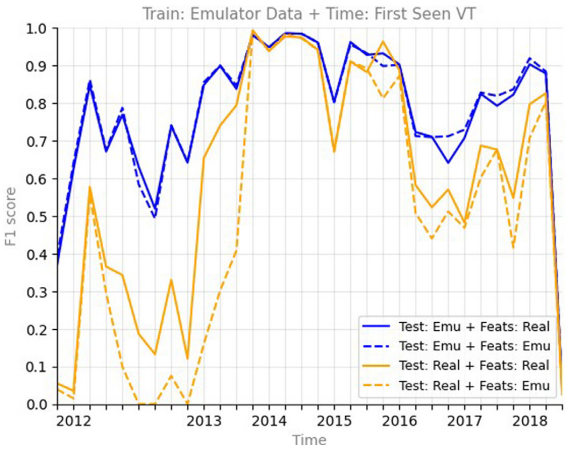


Fig. 8. First Seen timestamp - Training Emulator.

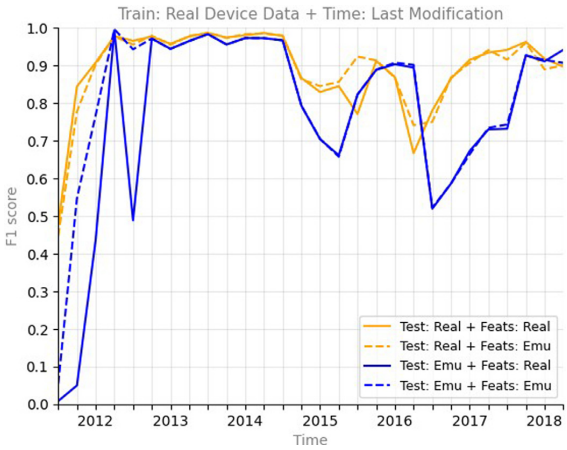


Fig. 7. Last Modification timestamp - Training Real Device.

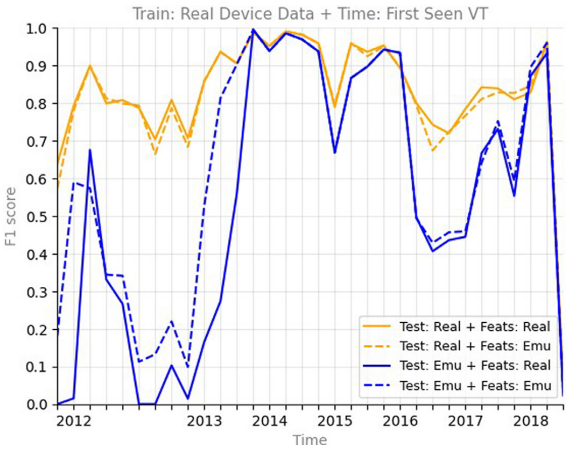


Fig. 9. First Seen timestamp - Training Real Device.

when the test data used does not belong to the same source as the training data. From 2016-Q3, a general worsening of the results is observed for all models, being especially pronounced when training and testing data sources differ. This fact is observed regardless of the feature set used. Overall, from the analysis of Figs. 6 and 7, it can be extracted that for the models built using the last modification timestamp, the feature set used does not have a relevant impact on the performance of the model, whereas the data source used has a significant decreasing impact on the model's outcome, especially if the training and testing set sources differ.

When the first seen timestamp is used to locate the same set of apps within the Android historical timeline instead of the last modification timestamp, the performance of the model is radically different as evidenced by Figs. 8 and 9. In this case, except for the period located between mid-2013 and the beginning of 2016 where the results are similar, the usage of distinct train and test sources yields very poor results. This fact is especially evident in the 2011–2013 time frame where the test detection performance using data distinct from the training source drops to null levels in two consecutive data chunks. Furthermore, the performance of the different feature sets allows the observer to draw contradictory conclusions in the first periods. For instance, as can be observed in Fig. 8, when the model is trained with emulator data and emulator-related features (i.e., reduced feature set) are used to describe the data, the testing with real device data provides worse performance than when real device-related features are used as descriptors (i.e., extended data set). However, the opposite situation is observed in Fig. 9 for the real device data trained models. In this case, when the extended feature set is used, testing with emulator data provides worse results than when the reduced feature set is used with the same testing data. This suggests that better results can be achieved when the model is trained with the *natural* feature set used to describe the testing source (i.e., the extended for real device data and the reduced for emulator data). Nevertheless, due to the aforementioned *zero-filling* imputation procedure for missing values when using the real device feature set to characterize emulator data, bias may be introduced to the model, and consequently, without proper knowledge about the source of

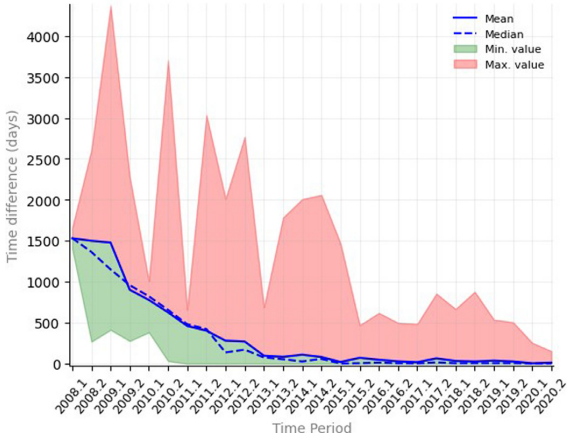


Fig. 10. Temporal differences between timestamps - Benign data.

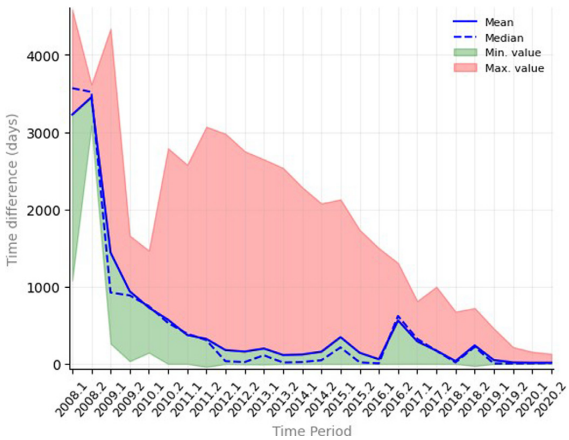


Fig. 11. Temporal differences between timestamps - Malware data.

the testing data, it is safer to use the reduced feature vector in all cases. The overall performance of the models built using the first seen timestamp is significantly worse than when the last modification timestamp is used. The overall performance is lower, especially when testing with a distinct data source, and the feature sets used seem to have a remarkable impact in the models. Furthermore, the performance varies abruptly from quarter to quarter indicating that sudden data drift occurs. This sudden drift could be caused by an *artificial* drift generated by the misplacement and the mix of *historically incoherent* data. Historical incoherence occurs when data belonging to different time-frames are blended together, thus generating an unnaturally occurring set of data. This is opposed to the overall smooth performance lines observed in the last modification timestamp, which may indicate a more naturally occurring drift in the data.

On the basis of the differences observed in the previous graphs, a deeper exploration of the timestamps becomes of interest. Every data sample can be located historically using either of the timestamps. The timestamps might converge or diverge. Timestamps converge if they provide similar timestamps, for instance, 2018 in both cases, whereas they diverge when their locations are significantly apart, for instance, one locates the sample in 2018 and the other in 2012. The temporal differences (i.e., divergence) between the timestamps can provide relevant insights for timestamp selection and analysis. Figs. 10 and 11 provide the temporal differences between both timestamps, computed for each sample and displaying benign and malware data separately. The individual differences in timestamps are grouped in six-month periods and located in the timeline based on the sample last modification timestamp, which is taken as reference. The individual differences for every period are averaged and reported in Figs. 10 and 11 for benign and malware samples, respectively. *Days* are used as the *difference* basic unit. More precisely, these graphs report the average value of the difference between the last modification timestamp and the first seen timestamp for the samples located in a specific period by the last modification timestamp. The initial expectation is that the last modification timestamp would place the sample more accurately in Android history (i.e., if not tampered), and earlier in time than the first seen timestamp. Therefore, it is chosen as the reference time. In Figs. 10 and 11, the blue line provides the average value for each period while the dashed line provides the median value. These two central tendency measures provide insights into the expected value of displacement of the samples for each period. The red and green areas provide the notion of the differences dispersion. More specifically, it is the range between the largest and the smallest difference found in that period (i.e., the maximum and minimum difference found in specific samples belonging to that chunk). The red area encompasses from the average value to the maximum value for each specific period, whereas the green area ranges from the average to the minimum value.

For both classes and all cases, a positive difference between both the timestamps is observed. This evidences that the first seen timestamp locates the samples later in time (i.e., delayed with respect to the last modification timestamp). The differences are especially pronounced in the early years of Android history, where differences average around 1500 days (i.e., four years) for benign applications and around 3500 days (i.e., over nine years) in the case of malware samples. For instance, a malware sample located in 2008.1 according to the last modification timestamp would be located in 2017.1 by the first seen timestamp (i.e., when VirusTotal first received the sample). This significant difference notably impacts the performance of the classifier and its adaptation to malware evolution when the concept drift issue is considered, as can be observed in Section 4.3.

However, as can be spotted in Figs. 10 and 11, these temporal differences have decreased over time. More precisely, they have monotonically decreased for benign instances and decreased significantly in the case of malware samples, thus making these timestamps more synchronized and closer in time. For example, for benign samples, 2020.1 and 2020.2 periods show a temporal difference average of just 4.88 and 12.37 days and a median of 2 and 3 days, respectively. In the case of malware samples, an average value of 15.98 and 16.45 days and 7 and 11 days are observed, respectively. As a result, the gap between both timestamp approaches has largely decreased over time, making them converge and increasing the reliability and accuracy of the first seen timestamp in the more recent years (i.e., 2019–2020).

The large differences observed in the early years of Android, with data misplaced about 4–8 years on average, have a significant impact on the models induced using the first seen timestamp, as evidenced in Figs. 8 and 9. In both cases, these initial periods provide the worst performance of all models induced for all tests cases, including when the test data source is the same as the training data source. Overall, as differences between timestamps decrease, thus first seen converges to the last modification

timestamp, the detection performance appears to improve, especially with regards to the test data from the same source as the train data. Therefore, it can be deduced that performance differences in the most recent years are mainly caused by apps' behavioral differences in different Android platforms rather than the impact of the timestamp, as the variation of the timestamp values minimizes.

The main derivation from these experiments indicates that the timestamp approach has a significant effect on the performance of concept drift models, especially for the first years of Android history. The last modification timestamp emerges as a more reliable temporal approximation to tackle the concept drift phenomenon, as evidenced by the high performance kept by the models in Figs. 6 and 7 over time and on both testing sets. Furthermore, the smooth transition between quarterly performances suggests that the data drift occurs more *naturally* and that the classifiers built on previous data chunks can be leveraged to accurately address emerging concept drift issues. The first seen timestamp performance transitions between adjacent quarters are abrupt and even extreme, thus indicating the existence of an *artificial* drift that can be hardly modeled using previous data knowledge.

Therefore, the last modification timestamp was selected to further explore the behavioral differences of Android apps across devices in the next section. Besides, data samples were described using the reduced feature set, which seems a more reasonable option for the selected timestamp.

4.4. Concept drift characterization

To perform a deeper analysis of the behavioral differences of the same set of apps in an emulator and a real device, *permutation feature importance* was calculated using *specificity* and *recall* as quality (Q) functions. In this exploration, for each analyzed case, the training and testing sources belonged to the same Android platform, thus enabling to describe the specific behavioral patterns for each device.

Feature importance calculated for *specificity* informs about the essential features to recognize benign software. The same function calculated for *recall* identifies the important features for malware recognition.

The graphs in Fig. 12 show the features with positive average feature importance for the benign and malware recognition task (i.e., specificity and recall). The permutation feature importance technique was calculated using 2000 permutations, and the experiment was repeated three times to ensure the stability of the results. For the real device data, 48 features were found important in at least one quarter. For the emulator, this number reached 65.

For each plot in Fig. 12, the colored lines/areas provide the important features for the related task (i.e., vertical axis) on each specific quarter (i.e., horizontal axis). The line/area color relates to specific features while the area width provides the *relative* feature importance of the specific feature. The relative importance of each system call on each specific chunk is reported in relative standing to the total importance of all the important system calls for that chunk. The relative measure of importance is preferred to the absolute importance value to provide a more comprehensive visual comparison of the important features over time. For the same reason, all features that showed a maximum relative importance value lower than five percent for any chunk in the whole analyzed period are aggregated in the *others* category. For each graph, the number of features aggregated in this category is provided within parentheses.

Fig. 12 a and b report on the important features for the benign software recognition task per quarter. As can be observed, a similar set of features are important on both devices for specificity performance. However, the proportion of features importance (i.e., relative importance) differs across devices. For instance, in the case of the emulator, the importance of *faccessat* feature is significant, especially in the first quarter, whereas, for the real device case, the importance is not remarkable in any period, thus it is included in the *others* category. On the other hand, *clock_gettime* and *read* have a remarkably greater importance for the real device than for the emulator. Besides, the first three periods for the emulator are described by a smaller set of important features, as evidenced by the thin lines and gaps observed in the importance plots. This might be a reason for the poor performance provided by the classifier in the initial periods when testing with distinct device data (see Fig. 7). Overall, similar sets are used but with varying proportions which makes the characterization substantially different. Furthermore, even though in the real device, a large set of features show importance, some of them show remarkably more importance than the others in the same chunk (e.g., *clock_gettime* in 2016–2017 time-frame and *recvfrom* in the 2014–2018 time frame) whereas, in the emulator, the importance is split among a larger number of features showing small individual importance values (with some exceptions like *recvfrom*).

Therefore, the usage of data from distinct sources as learning and testing sets for the benign recognition task may be biased and yield sub-optimal results but should be a relatively successful task due to the similar features describing effectively both sets.

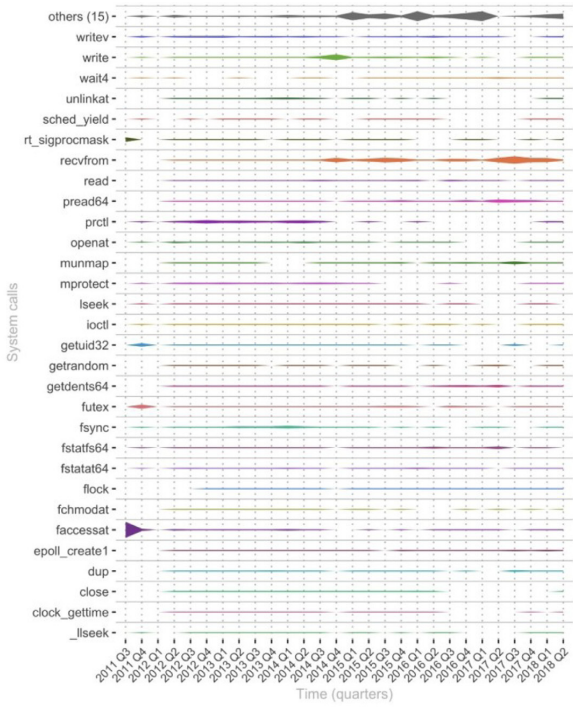
In order to analyze deeper the differences between the emulator and real device data, the obtained feature importance, calculated for specificity as Q function, were compared in each quarter for the emulator and the real device using the *Wilcoxon signed-rank* test. To apply this statistical test, the same number of values must be present in the compared sets. However, some features noted missing values in quarters where they were not found as an important feature for the task. Due to the high ratio of missing values in the analyzed data: 64% and 68% for the emulator and the real device, respectively, the imputation of zero value might bias the comparison results, increasing the similarity of the vectors due to the high number of missing values replaced by zero. A better approach was to replace the missing values with the mean importance of the feature, calculated for the given data set and taken as the reference *negative* value for the test. Thus, vectors were compared using means when the number of missing values was high and using the distribution of importance otherwise. The last issue was the different sets of important features for distinct data platforms. To avoid future complications, the intersection of both feature sets was used. As a result, 45 features were compared.

Table 6 (see Appendix) summarizes the experimental results. The table presents features with a p-value < 0.005 which suggests a great difference between the compared vectors. The occurrence value refers to the number of non-missing values in the compared vectors. As a result, only 13 features among 45 are significantly different for the emulated and real data. These results confirm the relative similarity observed in the distributions in Fig. 12a and b.

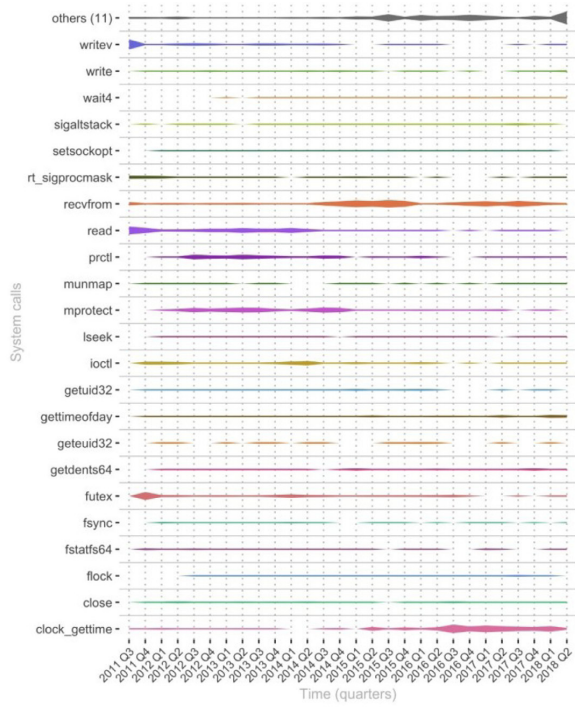
The situation changes drastically when the malicious software recognition task is analyzed.

Fig. 12 c and d show the set of features with positive average feature importance calculated for the *recall* metric, in the same manner as it was performed for specificity.

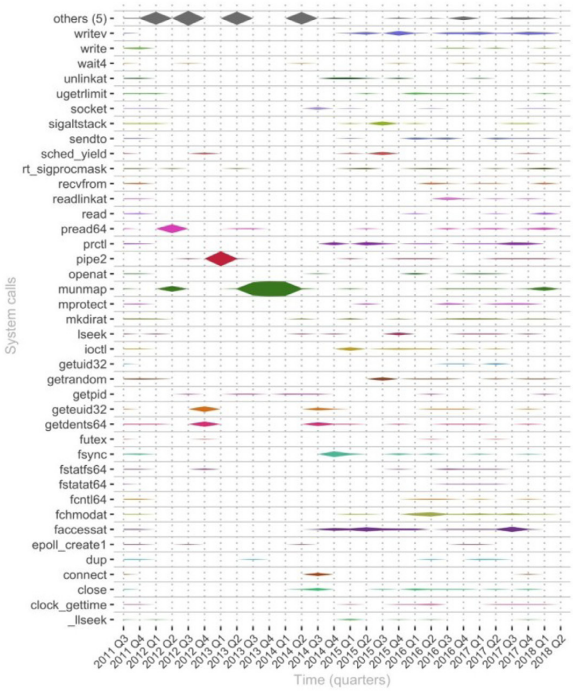
As can be observed, in both cases, from 2011 to 2016-Q2, the importance of the features calculated for recall differs significantly from the results obtained for specificity. More precisely, the incidental spikes of feature importance for recall show that a reduced set of features are important in each quarter, contrary to the large set of features observed of specificity. For the malware recognition task, in most of the quarters until 2016, less than 10 features emerge as important. However, this situation changed over time. In the last periods, the bars become more similar, as a larger amount of features become important for recall, thus the plots for



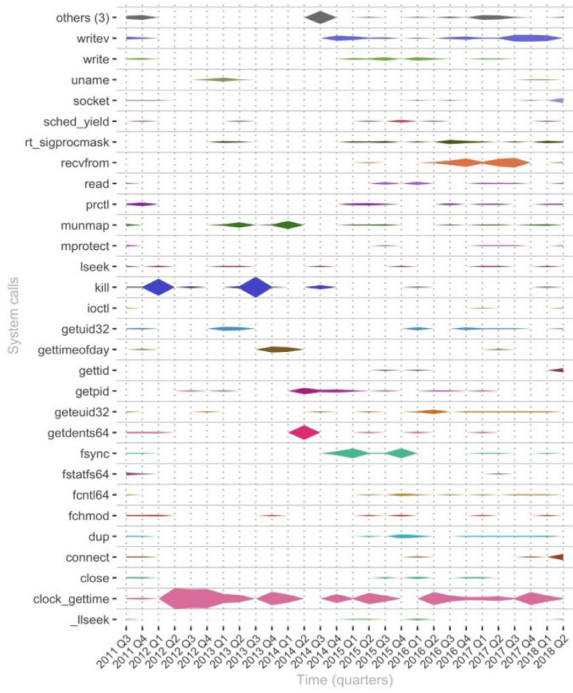
(a) Specificity - Emulator



(b) Specificity - Real Device



(c) Recall - Emulator



(d) Recall - Real Device

Fig. 12. Relative feature importance evolution from 2011-Q3 to 2018-Q2.

both recognition tasks resemble each other but still keep identifiable differences. However, even though these similarities are *strong* enough to relate the distinct plots according to devices, the global picture is that malware recognition is a completely different task than benign software recognition, thus relying on different sets of important features.

Although Fig. 12c and d show the same set of important features, there are essential differences between them related to importance levels. For instance, the most important feature is different for both sets. Real device data are dominated by *clock_gettime* feature, which is the most important feature in thirteen periods. However, in the emulator data, this feature is not present in fourteen periods, and it is never the most important feature in a quarter. For the emulator data, the most important feature is *munmap* but with a lower dominance than the most important feature in the real device case. Besides, a visual comparison between both graphs enables the observer to easily spot the differences in important features between these sets. Their character is radically distinct in the vast majority of the chunks. Therefore, it can be concluded that the usage of the same set of features can not guarantee the same results for both sets as the proportions and dominance of important features vary significantly.

The observed differences in importance influence cross-device malware detection quality. For instance, in each quarter that the classification F1 performance drops (see Fig. 7, training with real device data and testing with emulator), the feature *clock_gettime* is not considered as important in the emulator data. Besides, it is worth mentioning that the emulator data promotes *mprotect* feature, which is not significant in real data. In the timeframe from 2016.Q3 to 2017.Q3, when differences between F1 performance calculated for real device and emulated testing set are the largest, even when the same set of features is used, *recvfrom* feature is critical for recall. This feature is irrelevant for the emulator data in the same period.

Analogously to the specificity case, Table 7 (see Appendix) presents the *Wilcoxon signed-rank* test results for recall calculated between the emulator and real device data. In this case, over 75% of features (i.e., 34 out of 45) statistically differ between data sources (i.e., p-value < 0.005). Thus significant differences in features are thrice more prevalent for recall than for the specificity metric. The experimental results backed by the statistical analysis prove that emulator data differ significantly from real device data and that the coincident usage of both data sets is not advisable in a malware detection task.

As the previously presented *Wilcoxon signed-rank* tests performed for specificity and recall are remarkably different (i.e., in the number of features with different distribution), we compared the *importances* calculated separately for emulator and real device data using various quality functions Q.

Table 8 (see Appendix) provides the results of the comparison between recall and specificity calculated just with emulator data. The results obtained for the real device data are presented in Table 9 (see Appendix).

In both cases, the test results confirm the high number of features that differ in importance for the specificity and recall metrics. The number of different features exceeds 50% for the emulator data (i.e., 34 from 65 features) and 75% for the real device data (i.e., 35 from 45 features). These results support the previous observation on the difference between importance distributions. Furthermore, these results prove that benign software and malware develop in a disparate way and are characterized by different features. More importantly, the only observed similarities between the emulator and real device data in this sense are restricted to the fact that just 17 features are common in both tables, which is about half of the listed features. For instance, the most important features are not shared. The shared features are emphasized in italics in Tables 8 and 9 (see Appendix).

To summarize the findings from the previous tests, it could be stated that there exist significant behavioral cross-device differences when using system calls. Furthermore, these differences also exist for the classes within the same data source, as malware and benign data are distinctly characterized. Thus, a description of malware based on data from one device (i.e., real or emulated) should be used with much care to model classification for another device because of the significant differences in their characterization. This fact was also evidenced by the performance of the proposed models when the test sets used were collected from a distinct source than the train sets used to induce the models. Consequently, both data sources should not be merged in the generation and evaluation of Android malware detection methods.

5. Discussion

The previous concept drift modeling works in the mobile malware domain did not draw attention to the timestamp selection approach and its impact on the detection model performance. Our work shows that accurately grasping the evolution of malware and legitimate samples, thus effective modeling of the concept drift, greatly depends on the timestamp approach used, as the timestamp value is the main determinant in positioning the instance in the suitable chunk.

Concept drift detection results given in Section 4.2 (i.e., Figs. 4 and 5) indicate that the first seen timestamp creates more and deeper performance *blips* during the evolution of malware regardless of the data source (i.e., real device or emulator). The results of our concept drift model given in Section 4.3 support the same fact, especially in the early years of the analysis period, as *first seen* led to very poor detection performance (i.e., Figs. 8 and 9). Thus, we determined that the last modification timestamp better enables the model to reflect the changes and evolution of the threat landscape.

The dynamic behavior of the same set of Android apps is compared between distinct Android platforms (i.e., an emulator and a real device), providing experimental and statistical evidence of the changing behavior of apps according to the execution device. Our findings suggest that data from different sources should not be merged in training and testing sets as the behavior of apps in different Android platforms is not consistent.

The operational implications of hybrid detection systems, where cloud data might be merged with data acquired in users' devices, are critical. If the model is trained in a cloud-based *backend* with data collected from emulators and deployed to users' devices, then the detection performance might be significantly hindered as demonstrated in Figs. 6 and 8. As evidenced by Figs. 7 and 9, the best scenario for detection performance requires that train and test data are collected from the same Android platform, which in this case implies using either only real devices or only emulators as collection devices. Real device data collection is more time-consuming than the collection on emulators, as the devices have to be manually reset and *root*, and the collection cannot leverage *snapshots* for cleaning and restarting. This could make the process of building an effective model significantly longer. Furthermore, there may be further implications with the usage of distinct real devices, as due to the myriad of different devices, Android OS versions, and chipsets found in Android devices, behavioral dissimilarities may also arise among them. Thus impacting the performance of the detection system. However, it could be more comfortable for the user as it enables on-device detection. If the selection is only emulators, the user should send the app to be processed on the cloud, and after all the processing, the detection result should be

provided to the user. This implies the need for connectivity to retrieve the result, as the model is not deployed in the device and, consequently, a slower detection process. However, emulator models might be faster to induce and device parameters are easier to control, which may generate more robust models. As can be seen, the selection of any platform requires the analysis of pros and cons. In any case, the main recommendation is the usage of the same platform data for the training and test sets. Those solutions using mixed data sources should trace the data source and avoid merging the data when high detection performance is the objective.

Therefore, not only does the timestamp become a critical issue when concept drift is explored, but the source of data also emerges as an essential variable, becoming an even more significant issue as the timestamps converge. Both critical factors have been overlooked by the existing body of Android malware research. This study is the first step in further exploration of the critical impact of timestamps and devices in the data collection step.

This paper performs an attempt to characterize concept drift findings. We advocate for the incorporation of interpretation methods into application-oriented machine learning studies to have a better understanding of the underlying aspects of the problem domain beyond detection accuracy. Our proposal is not exclusively for explaining black-box models, but also researchers can discuss the findings that can be obtained from inherently *interpretable* models. This can enhance and enrich the knowledge of the problem domain. Some studies leverage the combination of machine learning and interpretability to explore a phenomenon (Stachl et al., 2019; Zhao et al., 2020). We are aware of the possible limitations of these methods (Molnar et al., 2020; Rudin, 2019), and that, usually, interpretability findings may not explain causality. Nevertheless, such an endeavor can increase the body of the domain knowledge that can be derived from the data. Beyond the purpose of knowledge generation, we also acknowledge that cyber security community should consider interpretability more as a part of the machine learning-based solutions due to the fact that security analysts play key role in security operations.

5.1. Threats to validity

This research aims to emphasize and bring to light several issues that have been overlooked by the existing research and may affect the field of Android malware detection. However, this research is not free of limitations, which are summarized as follows.

- The data used in this research is specifically tailored to analyze concept drift-related issues. The same data set is analyzed on both Android platforms but issues such as malware *anti-sandbox* capabilities are not taken into account. Nevertheless, other third variables which may generate differences in behavioral profiles were controlled in the generation of the data set (i.e., using the same OS version, scripts, and debugging tool).
- The Android platforms used to run the applications were selected on the basis of user preferences and device popularity (i.e., Google Emulator and a Samsung device) (Guerra-Manzanares et al., 2021). However as they may change and evolve over time, other devices could have been used which might have provided distinct results.
- The characterization technique (i.e., permutation feature importance) has been widely used to characterize machine learning models. Nevertheless, due to its inherent randomization procedure, it might be prone to show distinct pictures of models, especially when feature randomization is embedded into the models, such as in the Random Forest algorithm.

Therefore, even though this study has limitations, they have been tested and minimized to provide the most complete approach to the phenomenon.

6. Conclusions and future work

The vast majority of literature in Android malware detection has neglected the detrimental impact of the *time* variable in the machine learning-based detection models and has not even considered the implications of merging data sources. Furthermore, the small number of concept drift-related studies have not paid attention to the timestamp selection, a central element to address concept drift effectively.

This research explores the emerging challenges and their implications when dealing with concept drift for Android malware detection using different timestamps and, at the same time, using data collected from distinct Android platforms (i.e., real device and emulator). Our results show the detrimental impact on machine learning classifiers caused by concept drift, especially when using *first seen* as the timestamp. The *last modification* timestamp appears to be a reliable and accurate source for the historical location of apps in the Android timeline, providing and keeping high-performance detection models over time, even when distinct data sources are considered. However, our experimental setup proves that the behavior of Android apps is not consistent across Android platforms and that data collected from different sources should not be used coincidentally.

An extensive body of research has previously focused on the optimization of proposed solutions in short and mostly outdated Android historical data sets. The focus on these solutions tailored for static data snapshots poses severe concerns about the generalization capabilities of such solutions to recent malware. To the best of our knowledge, this research is the first work addressing the challenges and implications of distinct timestamps for historical location, concept drift issues, and cross-device data for Android malware detection. This work aims to bring to light the significant impact of these underlying critical variables that have been overlooked by the specialized research.

In our future work, we plan to continue digging deeper into this approach and tackle the aforementioned limitations in a more restrictive way. This work should be understood as a pioneering step into a new exploratory direction that considers the impact of several variables on the performance of the learning models and challenges the assumptions of most of the previous research.

Declaration of Competing Interest

We wish to confirm that there are no known conflicts of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

CRediT authorship contribution statement

Alejandro Guerra-Manzanares: Conceptualization, Methodology, Software, Validation, Investigation, Visualization, Writing – original draft, Writing – review & editing. **Marcin Luckner:** Conceptualization, Methodology, Investigation, Visualization, Writing – original draft, Writing – review & editing. **Hayretdin Bahsi:** Conceptualization, Methodology, Writing – review & editing.

Appendix A

Table 6
Features that differ between emulated and real device data in importance calculated for Specificity $p < 0.005$.

	Feature	*p*-value	max(Importance)	Occurrences
1	recvfrom	0.0036	0.2250	53
2	clock_gettime	0.0001	0.1423	43
3	setsockopt	0.0000	0.0755	39
4	uname	0.0000	0.0193	20
5	statfs64	0.0000	0.0218	17
6	exit_group	0.0000	0.0006	12
7	gettid	0.0001	0.0063	10
8	tgkill	0.0000	0.0001	6
9	nanosleep	0.0000	0.0000	3
10	sched_getscheduler	0.0000	0.0000	2
11	msync	0.0000	0.0141	2
12	listen	0.0000	0.0000	1
13	mremap	0.0000	0.0070	1

Table 7
Features that differ between emulated and real device data in importance calculated for Recall $p < 0.005$.

	Feature	*p*-value	max(Importance)	Occurrences
1	clock_gettime	0.0032	0.0825	32
2	getdents64	0.0002	0.0161	19
3	fsync	0.0011	0.0527	18
4	lseek	0.0050	0.0111	18
5	dup	0.0000	0.0375	17
6	fcntl64	0.0001	0.0477	16
7	socket	0.0001	0.1513	15
8	write	0.0000	0.0197	15
9	recvfrom	0.0016	0.0728	15
10	fchmod	0.0000	0.0058	14
11	getuid32	0.0000	0.0363	14
12	mprotect	0.0015	0.0484	14
13	sigaltstack	0.0000	0.0077	14
14	read	0.0000	0.0313	12
15	_llseek	0.0000	0.0263	11
16	fstatfs64	0.0000	0.0710	11
17	sched_yield	0.0000	0.0063	11
18	sendmsg	0.0000	0.0071	9
19	setsockopt	0.0000	0.0058	9
20	connect	0.0000	0.1851	8
21	futex	0.0000	0.0441	8
22	uname	0.0000	0.0018	6
23	wait4	0.0000	0.0059	6
24	exit_group	0.0000	0.0001	5
25	getpriority	0.0000	0.0022	5
26	flock	0.0000	0.0059	4
27	listen	0.0000	0.0000	4
28	getcwd	0.0000	0.0000	3
29	gettid	0.0000	0.1234	3
30	restart_syscall	0.0000	0.0000	2
31	tgkill	0.0000	0.0000	2
32	nanosleep	0.0000	0.0000	2
33	mremap	0.0000	0.0000	2
34	msync	0.0000	0.0000	2

Table 8
Features with different importance for Specificity and Recall for emulated data $p < 0.005$.

	Feature	*p*-value	max(Importance)	Occurrences
1	writev	0.0038	0.1808	35
2	*getdents64*	0.0003	0.0303	34
3	pread64	0.0001	0.0587	34
4	*recvfrom*	0.0000	0.1070	32
5	fstatfs64	0.0002	0.0545	31
6	epoll_create1	0.0000	0.0702	30
7	fcntl64	0.0009	0.0556	30
8	*lseek*	0.0028	0.0548	30
9	*fstatat64*	0.0000	0.1219	29
10	*write*	0.0002	0.1536	29
11	*futex*	0.0000	0.0696	28
12	*read*	0.0005	0.0994	28
13	dup	0.0039	0.0724	27
14	eventfd2	0.0011	0.0304	27
15	*flock*	0.0005	0.0189	26
16	*setsockopt*	0.0000	0.0058	19
17	pipe2	0.0001	0.0069	18
18	statfs64	0.0000	0.0090	17
19	process_vm_readv	0.0006	0.0001	12
20	restart_syscall	0.0000	0.0003	11
21	inotify_add_watch	0.0000	0.0001	7
22	ppoll	0.0000	0.0045	6
23	*exit_group*	0.0000	0.0006	6
24	*tgkill*	0.0000	0.0001	5
25	*nanosleep*	0.0000	0.0000	4
26	inotify_init1	0.0000	0.0001	4
27	msync	0.0000	0.0141	4
28	*gettid*	0.0000	0.0001	4
29	*mremap*	0.0000	0.0070	2
30	uname	0.0000	0.0000	1
31	*sched_getscheduler*	0.0000	0.0000	1
32	*listen*	0.0000	0.0000	1
33	*getcwd*	0.0000	0.0000	1
34	rt_sigsuspend	0.0000	0.0000	1

Table 9
Features with different importance for Specificity and Recall for real device data $p < 0.005$.

	Feature	*p*-value	max(Importance)	Occurrences
1	clock_gettime	0.0019	0.1423	47
2	*recvfrom*	0.0000	0.2250	36
3	rt_sigprocmask	0.0047	0.1716	34
4	*lseek*	0.0000	0.0550	33
5	prctl	0.0007	0.3801	33
6	close	0.0000	0.1269	32
7	*getdents64*	0.0000	0.0601	32
8	getuid32	0.0013	0.0751	32
9	*read*	0.0000	0.4032	32
10	*write*	0.0001	0.1145	32
11	connect	0.0000	0.1851	31
12	gettimeofday	0.0000	0.0755	31
13	mprotect	0.0000	0.3975	30
14	sigaltstack	0.0000	0.0265	30
15	sched_yield	0.0000	0.0248	30
16	*setsockopt*	0.0000	0.0755	29
17	*futex*	0.0000	0.2334	28
18	ioctl	0.0000	0.2304	28
19	socket	0.0000	0.1513	27
20	getpid	0.0014	0.0081	27

(*continued on next page*)

Table 9 (*continued*)

	Feature	*p*-value	max(Importance)	Occurrences
21	getpriority	0.0000	0.0519	27
22	sendmsg	0.0005	0.0432	26
23	*fstatfs64*	0.0004	0.0990	25
24	*flock*	0.0000	0.0371	23
25	wait4	0.0000	0.0220	21
26	*exit_group*	0.0001	0.0003	11
27	*gettid*	0.0000	0.1234	9
28	getegid32	0.0000	0.0002	6
29	getgid32	0.0000	0.0001	5
30	*listen*	0.0000	0.0000	4
31	*tgkill*	0.0000	0.0000	3
32	*getcwd*	0.0000	0.0000	2
33	*sched_getscheduler*	0.0000	0.0000	1
34	*nanosleep*	0.0000	0.0000	1
35	*mremap*	0.0000	0.0000	1

References

Aggarwal, C.C., 2015. Data Mining: The Textbook. Springer.

Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y., 2015. Are your training datasets yet relevant? In: International Symposium on Engineering Secure Software and Systems. Springer, pp. 51–67.

Altmann, A., Toloşi, L., Sander, O., Lengauer, T., et al., 2010. Permutation importance: a corrected feature importance measure. Bioinformatics 26 (10), 1340–1347. doi:10.1093/bioinformatics/btq134.

Alzaylaee, M.K., Yerima, S.Y., Sezer, S., 2017. Emulator vs. real phone: android malware detection using machine learning. In: Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics, pp. 65–72.

Alzaylaee, M.K., Yerima, S.Y., Sezer, S., 2020. DL-Droid: deep learning based android malware detection using real devices. Comput. Secur. 89, 101663.

Amin, M.R., Zaman, M., Hossain, M.S., Atiquzzaman, M., 2016. Behavioral malware detection approaches for android. In: 2016 IEEE International Conference on Communications (ICC), pp. 1–6. doi:10.1109/ICC.2016.7511573.

Android. Run apps on the android emulator. https://developer.android.com/studio/run/emulator; 2021.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C., 2014. DREBIN: effective and explainable detection of android malware in your pocket. In: Ndss, vol. 14, pp. 23–26.

Barbero F., Pendlebury F., Pierazzi F., Cavallaro L.. Transcending transcend: revisiting malware classification with conformal evaluation. arXiv preprint arXiv:201003856 2020.

Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32. doi:10.1023/A:1010933404324.

Burguera, I., Zurutuza, U., Nadjm-Tehrani, S., 2011. Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, pp. 15–26.

Cai, H., 2020. Assessing and improving malware detection sustainability through app evolution studies. ACM Trans. Softw. Eng. Methodol. (TOSEM) 29 (2), 1–28.

Cai, H., Meng, N., Ryder, B., Yao, D., 2019. DroidCat: effective android malware detection and categorization via app-level profiling. IEEE Trans. Inf. Forensics Secur. 14 (6), 1455–1470. doi:10.1109/TIFS.2018.2879302.

Cai, L., Li, Y., Xiong, Z., 2021. Jowmdroid: android malware detection based on feature weighting with joint optimization of weight-mapping and classifier parameters. Comput. Secur. 100, 102086.

Casolare, R., De Dominicis, C., Iadarola, G., Martinelli, F., Mercaldo, F., Santone, A., 2021. Dynamic mobile malware detection through system call-based image representation. J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl. 12 (1), 44–63.

Dimjašević, M., Atzeni, S., Ugrina, I., Rakamaric, Z., 2016. Evaluation of android malware detection based on system calls. In: Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics, pp. 1–8.

Fedler, R., Schütte, J., Kulicke, M., 2013. On the effectiveness of malware protection on android. Fraunhofer AISEC 45.

Feng, P., Ma, J., Sun, C., Xu, X., Ma, Y., 2018. A novel dynamic android malware detection system with ensemble learning. IEEE Access 6, 30996–31011. doi:10.1109/ACCESS.2018.2844349.

Gao, H., Cheng, S., Zhang, W., 2021. Gdroid: android malware detection and classification with graph convolutional network. Comput. Secur. 106, 102264.

Google. Google play protect. https://developers.google.com/android/play-protect; 2021.

Gözüaçk, O., Can, F., 2020. Concept learning using one-class classifiers for implicit drift detection in evolving data streams. Artif. Intell. Rev. (0123456789) doi:10.1007/s10462-020-09939-x.

Guerra-Manzanares, A., Bahsi, H., Nõmm, S., 2021. Kronodroid: time-based hybrid-featured dataset for effective android malware detection and characterization. Comput. Secur. 102399.

Guerra-Manzanares, A., Bahsi, H., Nõmm, S., 2019a. Differences in android behavior between real device and emulator: a malware detection perspective. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), pp. 399–404. doi:10.1109/IOTSMS48152.2019.8939268.

Guerra-Manzanares, A., Luckner, M., Bahsi, H., 2022. Android malware concept drift using system calls. Under Rev.

Guerra-Manzanares, A., Nõmm, S., Bahsi, H., 2019b. Time-frame analysis of system calls behavior in machine learning-based mobile malware detection. In: 2019 International Conference on Cyber Security for Emerging Technologies (CSET). IEEE, pp. 1–8.

Guerra-Manzanares, A., Nomm, S., Bahsi, H., 2019c. In-depth feature selection and ranking for automated detection of mobile malware. In: ICISSP, pp. 274–283.

Han, Q., Subrahmanian, V.S., Xiong, Y., 2020. Android malware detection via (somewhat) robust irreversible feature transformations. IEEE Trans. Inf. Forensics Secur. 15, 3511–3525. doi:10.1109/TIFS.2020.2975932.

Iadarola, G., Martinelli, F., Mercaldo, F., Santone, A., et al., 2021. Towards an interpretable deep learning model for mobile malware detection and family identification. Comput. Secur. 105, 102198. doi:10.1016/j.cose.2021.102198.

Irolla, P., Dey, A., 2018. The duplication issue within the Drebin dataset. J. Comput. Virol. Hack. Tech. 14 (3), 245–249.

Jerbi, M., Dagdia, Z.C., Bechikh, S., Said, L.B., 2020. On the use of artificial malicious patterns for android malware detection. Comput. Secur. 92, 101743.

Jordaney, R., Sharad, K., Dash, S.K., Wang, Z., Papini, D., Nouretdinov, I., Cavallaro, L., 2017. Transcend: detecting concept drift in malware classification models. In: 26th {USENIX} Security Symposium ({USENIX} Security 17), pp. 625–642.

Karn, R.R., Kudva, P., Huang, H., Suneja, S., Elfadel, I.M., et al., 2021. Cryptomining detection in container clouds using system calls and explainable machine learning. IEEE Trans. Parallel Distrib. Syst. 32 (3), 674–691. doi:10.1109/TPDS.2020.3029088.

Kaspersky. Mobile security: Android vs. iOS - which one is safer?https://www.kaspersky.com/resource-center/threats/android-vs-iphone-mobile-security; 2020.

Kinkead, M., Millar, S., McLaughlin, N., O'Kane, P., et al., 2021. Towards explainable CNNs for android malware detection. Procedia Comput. Sci. 184 (2019), 959–965. doi:10.1016/j.procs.2021.03.118.

Lei, T., Qin, Z., Wang, Z., Li, Q., Ye, D., 2019. Evedroid: event-aware android malware detection against model degrading for IoT devices. IEEE Internet Things J. 6 (4), 6668–6680. doi:10.1109/JIOT.2019.2909745.

Lin, Y.D., Lai, Y.C., Chen, C.H., Tsai, H.C., 2013. Identifying android malicious repackaged applications by thread-grained system call sequences. Comput. Secur. 39, 340–350.

Lindorfer, M., Neugschwandtner, M., Platzer, C., 2015. MARVIN: efficient and comprehensive mobile app classification through static and dynamic analysis. In: 2015 IEEE 39th Annual Computer Software and Applications Conference, vol. 2. IEEE, pp. 422–433.

Liu, K., Xu, S., Xu, G., Zhang, M., Sun, D., Liu, H., 2020. A review of android malware detection approaches based on machine learning. IEEE Access 8, 124579–124607.

Lu, N., Zhang, G., Lu, J., 2014. Concept drift detection via competence models. Artif. Intell. 209 (1), 11–28. doi:10.1016/j.artint.2014.01.001.

U. du Luxembourg. Androzoo - lists of APKs. https://androzoo.uni.lu/lists; 2021.

, 2005. In: Maimon, O., Rokach, L. (Eds.), Data Mining and Knowledge Discovery Handbook. A Complete Guide for Practitioners and Researchers. Springer, San Francisco, CA, USA.

Margara A., Rabl T.. Definition of Data Streams; Cham: Springer International Publishing. p. 1–4. doi:10.1007/978-3-319-63962-8_188-1.

Molnar C., König G., Herbinger J., Freiesleben T., Dandl S., Scholbeck C.A., Casalicchio G., Grosse-Wentrup M., Bischl B.. Pitfalls to avoid when interpreting machine learning models. arXiv preprint arXiv:200704131 2020.

Mutz, D., Valeur, F., Vigna, G., Kruegel, C., 2006. Anomalous system call detection. ACM Trans. Inf. Syst. Secur. 9 (1), 61–93. doi:10.1145/1127345.1127348.

Narayanan, A., Yang, L., Chen, L., Jinliang, L., 2016. Adaptive and scalable android malware detection through online learning. In: 2016 International Joint Conference on Neural Networks (IJCNN), pp. 2484–2491. doi:10.1109/IJCNN.2016.7727508.

Naval, S., Laxmi, V., Rajarajan, M., Gaur, M.S., Conti, M., 2015. Employing program semantics for malware detection. IEEE Trans. Inf. Forensics Secur. 10 (12), 2591–2604. doi:10.1109/TIFS.2015.2469253.

Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G., 2019. Mamadroid: detecting android malware by building Markov chains of behavioral models (extended version). ACM Trans. Privacy Secur. (TOPS) 22 (2), 1–34.

Palmer D.. Sophisticated android malware spies on smartphones users and runs up their phone bill too. https://www.zdnet.com/article/sophisticated-android-malware-spies-on-smartphones-users-and-runs-up-their-phone-bill-too/; 2018.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L., 2019. {TESSERACT}: eliminating experimental bias in malware classification across space and time. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 729–746.

Ramírez-Gallego, S., Krawczyk, B., García, S., Woźniak, M., Herrera, F., et al., 2017. A survey on data preprocessing for data stream mining: current status and future directions. Neurocomputing 239, 39–57. doi:10.1016/j.neucom.2017.01.078.

Rudin, C., 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. Nat. Mach. Intell. 1 (5), 206–215.

Ruiz-Heras, A., García-Teodoro, P., Sánchez-Casado, L., 2017. ADroid: Anomaly-based detection of malicious events in android platforms. Int. J. Inf. Secur. 16 (4), 371–384. doi:10.1007/s10207-016-0333-1.

Samsung. About knox. https://www.samsungknox.com/en/about-knox; 2021.
Saracino, A., Sgandurra, D., Dini, G., Martinelli, F., 2018. Madam: effective and efficient behavior-based android malware detection and prevention. IEEE Trans. Dependable Secure Comput. 15 (1), 83–97. doi:10.1109/TDSC.2016.2536605.
Sasidharan, S.K., Thomas, C., 2021. ProDroid—An android malware detection framework based on profile hidden Markov model. Pervasive Mob. Comput. 72, 101336.
Scalas, M., Maiorca, D., Mercaldo, F., Visaggio, C.A., Martinelli, F., Giacinto, G., et al., 2019. On the effectiveness of system API-related information for android ransomware detection. Comput. Secur. 86, 168–182. doi:10.1016/j.cose.2019.06.004.
Seiffert, C., Khoshgoftaar, T.M., Van Hulse, J., Napolitano, A., et al., 2010. RUSBoost: a hybrid approach to alleviating class imbalance. IEEE Trans. Syst., Man, Cybern. Part A 40 (1), 185–197. doi:10.1109/TSMCA.2009.2029559.
Sharma, T., Rattan, D., 2021. Malicious application detection in android—A systematic literature review. Comput. Sci. Rev. 40, 100373.
Sophos. Malware goes mobile: Timeline of mobile threats, 2004–2016. https://www.sophos.com/en-us/medialibrary/PDFs/marketing%20material/sophos-threat-infographic-ten-years-malware-mobile-devices.pdf; 2017.
Stachl C., Au Q., Schoedel R., Buschek D., Völkel S., Schuwerk T., Oldemeier M., Ullmann T., Hussmann H., Bischl B., et al. Behavioral patterns in smartphone usage predict big five personality traits2019;.
Statista. Mobile operating system market share worldwide, July 2020–July 2021. https://gs.statcounter.com/os-market-share/mobile/worldwide; 2021.
Surendran, R., Thomas, T., Emmanuel, S., 2020. Gsdroid: graph signal based compact feature representation for android malware detection. Expert Syst. Appl. 159, 113581.
Vidal, J.M., Orozco, A.L.S., Villalba, L.J.G., 2017. Malware detection in mobile devices by analyzing sequences of system calls. World Acad. Sci., Eng.Technol., Int. J. Comput., Electr., Autom., Control Inf. Eng. 11 (5), 594–598.
Vinod, P., Zemmari, A., Conti, M., 2019. A machine learning based approach to detect malicious android apps using discriminant system calls. Future Gener. Comput. Syst. 94, 333–350.
VirusTotal. An update from virustotal. https://blog.virustotal.com/2012/09/an-update-from-virustotal.html; 2012.
Wang, X., Li, C., 2021. Android malware detection through machine learning on kernel task structures. Neurocomputing 435, 126–150.
Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W., 2017. Deep ground truth analysis of current android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 252–276.
Wei, W., Wang, J., Yan, Z., Ding, W., 2022. EPMDroid: efficient and privacy-preserving malware detection based on SGX through data fusion. Inf. Fusion.
Whitwam R.. Android antivirus apps are useless - here's what to do instead. https://www.extremetech.com/computing/104827-android-antivirus-apps-are-useless-heres-what-to-do-instead; 2021.
Xiao, X., Zhang, S., Mercaldo, F., Hu, G., Sangaiah, A.K., 2019. Android malware detection based on system call sequences and LSTM. Multimed. Tools Appl. 78 (4), 3979–3999.
Xu, K., Li, Y., Deng, R., Chen, K., Xu, J., 2019. Droidevolver: self-evolving android malware detection system. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, pp. 47–62.
Yaswant A.. New advanced android malware posing as "system update". https://blog.zimperium.com/new-advanced-android-malware-posing-as-system-update/; 2021.
Zhang, N., Xue, J., Ma, Y., Zhang, R., Liang, T., Tan, Y.a., 2021. Hybrid sequence-based android malware detection using natural language processing. Int. J. Intell. Syst. 36 (10), 5770–5784.
Zhang, X., Zhang, Y., Zhong, M., Ding, D., Cao, Y., Zhang, Y., Zhang, M., Yang, M., 2020. Enhancing state-of-the-art classifiers with API semantics to detect evolved android malware. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 757–770.
Zhao, X., Lovreglio, R., Nilsson, D., 2020. Modelling and interpreting pre-evacuation decision-making using machine learning. Autom. Constr. 113, 103140.
Zhou, Y., Jiang, X., 2012. Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy, pp. 95–109. doi:10.1109/SP.2012.16.
Zyblewski, P., Sabourin, R., Woźniak, M., 2021. Preprocessed dynamic classifier ensemble selection for highly imbalanced drifted data streams. Inf. Fusion 66 (June 2020), 138–154. doi:10.1016/j.inffus.2020.09.004.

Alejandro Guerra Manzanares is a Ph.D. candidate at the Center for Digital Forensics and Cyber Security, Department of Software Science, Tallinn University of Technology (Estonia). He received a BA degree in criminology from the Autonomous University of Barcelona (Spain) in 2013, and a BS degree in ICT engineering from the Polytechnic University of Catalonia (Spain) in 2017. In 2018, he received a M.Sc. in cyber security from Tallinn University of Technology (Estonia). His research interests are in the application of machine learning techniques to digital forensics and cyber securityrelated issues, such as mobile malware and IoT botnet detection.

Marcin Luckner works at the Faculty of Mathematics and Information Science, Warsaw University of Technology. He received his Ph.D. degree from the System Research Institute of the Polish Academy of Science in 2010. His primary interests are in pattern recognition and artificial intelligence and their applications, as well as in anomaly detection. He has published about 40 papers. He has managed Research and Development projects for business on behalf of the academic research center, including several projects focused on cyber-security in cell networks and web-spam detection.

Hayretdin Bahsi is a research professor at the Center for Digital Forensics and Cyber Security at Tallinn University of Technology, Estonia. He has two decades of professional and academic experience in cybersecurity. He received his Ph.D. from Sabanc University (Turkey) in 2010. He was involved in many R&D and consultancy projects about cybersecurity as a researcher, consultant, trainer, project manager, and program coordinator at the National Cyber Security Research Institute of Turkey between 2000 and 2014. His research interests include machine learning and its application to cyber security and digital forensic problems.

Appendix 7

Publication VII
A. Guerra-Manzanares and M. Välbe. Cross-device behavioral consistency: Benchmarking and implications for effective android malware detection. *Machine Learning with Applications*, 9:100357, 2022

Machine Learning with Applications 9 (2022) 100357

Contents lists available at ScienceDirect

Machine Learning with Applications

journal homepage: www.elsevier.com/locate/mlwa

Cross-device behavioral consistency: Benchmarking and implications for effective android malware detection

Alejandro Guerra-Manzanares *, Martin Välbe

Department of Software Science, Tallinn University of Technology, Estonia

ARTICLE INFO

Keywords:
Benchmark
Android malware
Malware detection
Malware behavior
System calls
Real device
Android emulator

ABSTRACT

Most of the proposed solutions using dynamic features for Android malware detection collect and test their systems using a single and particular data collection device, either a real device or an emulator. The results obtained using these particular devices are then generalized to any Android platform. This extensive generalization is based on the assumption of consistent behavior of apps across devices. This study performs an extensive benchmarking of this assumption for system calls, executing Android malware and benign samples under the same conditions in 9 different collection devices, including real and virtual devices. The results indicate the existence of significant differences between real devices and emulators in system calls usage and, consequently, in the collected behavioral profiles obtained from running the same set of applications on different devices. Furthermore, the impact of these differences on machine learning-based malware detection models is evaluated. In this regard, a significant degenerative effect on the detection performance of the model is produced when data collected on different devices are used in the training and testing sets. Therefore, the empirical findings do not support the assumption of cross-device consistent behavior of Android apps when system calls are used as descriptive features.

1. Introduction

Android users are under siege. The open-source nature and ubiquity of the operating system (OS), which powers over 70% of smartphones (O'Dea, 2021) and more than 2.5 billion users worldwide (Curry, 2021), make it an appealing target for cyber attackers. The most *popular* mobile OS has been the objective of massive malware campaigns since its early days (Keizer, 2012; Sophos, 2017) and is still under a seemingly inexorable attack (Afifi-Sabet, 2019; CheckPoint, 2017; Islam, 2021). After the record-breaking spike of mobile malware in 2016–2017, Android malware attacks have persistently remained at high levels (Chebyshev, 2021b), posing a significant and constant threat to the end-users (AV-Test, 2021). For instance, more than 700,000 new malware instances were discovered monthly in the last quarter of 2020 and over 480,000 in the first quarter of 2021 (Chebyshev, 2021a; Statista, 2021).

Despite the remarkable efforts of Google and Android original equipment manufacturers (OEMs) to address the situation by implementing counter-means at the software (Google, 2021) and hardware (Thinagarajan, 2021) level, Android malware is still a very lucrative and cost-effective endeavor for attackers (Cimpanu, 2021; Ilascu, 2020). Malicious applications have been repeatedly found in Google Play (McGowan, 2020; Stokel-Walker, 2021) and *third-party* app stores thus no safe shelter against the malware spread seems to exist. Malicious apps disguise in many forms and shapes, a product of the constant evolution and adaptation in the seek for new vulnerabilities and attack vectors to perpetrate their obscure intentions (Microsoft, 2020; Yaswant, 2021). In this large threat landscape, Android end users are vulnerable prey, at permanent risk (Kingsley-Hughes, 2021).

Furthermore, the ineffectiveness and limitations of traditional signature-based detection methods on the mobile platform have not helped to remedy the situation (Fedler, Schutte, & Kulicke, 2013; Whitwam, 2021). In the search for solutions to balance forces in the permanent battle between malicious actors and defenders, machine learning techniques (ML) have been explored. In this regard, ML-based solutions have proved remarkably successful in detecting and neutralizing malicious threats, even against *zero-day* and *obfuscated* malware (Millar, McLaughlin, del Rincon, & Miller, 2021; Yerima, Sezer, & Muttik, 2014).

In supervised learning, machine learning-based malware detection systems *learn* patterns and statistical relationships from *known* data (i.e., *training* data) to make accurate forecasts about *unknown* data (i.e., *testing* data). The quantity, but more importantly, the quality

The code (and data) in this article has been certified as Reproducible by Code Ocean: (https://codeocean.com/). More information on the Reproducibility Badge Initiative is available at https://www.elsevier.com/physical-sciences-and-engineering/computer-science/journals.

* Corresponding author.
E-mail addresses: alejandro.guerra@taltech.ee (A. Guerra-Manzanares), mavalb@taltech.ee (M. Välbe).

https://doi.org/10.1016/j.mlwa.2022.100357
Received 25 January 2022; Received in revised form 6 June 2022; Accepted 7 June 2022
Available online 18 June 2022

of the training data are critical for the effectiveness of the detection system (Cortes, Jackel, Chiang, et al., 1995). Training data must be accurate and representative of the task that the ML model aims to solve (e.g., malware detection) so that the model can generalize effectively to unknown data, that is, to predict accurately *new* data.

For malware detection, *static* and *dynamic* data features are collected using malware analysis techniques and used as input data to induce ML-based models. Static feature are extracted directly from the file or source code without executing the app. In general, static features are relatively easy and fast to acquire. However, the detection systems built using only this type of features are prone to be bypassed when code obfuscation and encryption means are implemented (Lysne, 2018). The collection of dynamic features requires the execution of the app in a live environment to be collected. The acquisition of these features is more time-consuming, but they tend to build more robust detection systems as they are inherently immune to code obfuscation and encryption. Besides, they capture the real *behavior* of the app and are usually more effective to detect zero-day malware (Networks, 2021). In this regard, system calls are the most used dynamic feature for Android malware detection and the basis of many high-performance solutions in the related literature (Burguera, Zurutuza, & Nadjm-Tehrani, 2011; Dimjasevic, Atzeni, Ugrina, & Rakamaric, 2016; Guerra-Manzanares, Nõmm, & Bahsi, 2019b; Hou, Saas, Chen, & Ye, 2016). Research studies use either real devices or emulators as data collection platforms to acquire system call-related features. More commonly, the data samples are *run* in a single collection platform under a particular configuration (i.e., a real device or emulator running a specific version of Android OS). As a result, a myriad of different devices and configurations are used as acquisition platforms. After the data is processed, the collected data are fed into machine learning algorithms to build effective detection solutions that usually report high-performance metrics. The detection capabilities of the built systems and proposed solutions are assumed (but not proven) to generalize to the plethora of distinct devices and configurations found *in the wild* on the basis of an axiomatic *cross-device behavioral consistency*. Thus assuming that the behavior of apps is fully consistent across devices (Lin, Lai, Chen, & Tsai, 2013) and OS versions (Burguera et al., 2011; Vidal, Orozco, & Villalba, 2017). This fact, in turn, implies that the nature and characteristics of the devices (i.e., emulators or real devices) and Android versions used as collection platforms do not matter, as evidenced by the lack of homogeneity in selection criteria and the wide variety of devices and configurations found in research studies.

Nevertheless, the results of the reduced number of studies that considered both kinds of devices in their experimentation challenge the *validity* of this axiomatic consistency. More specifically, when different platforms were considered in the same study, the results did not support the *fully consistent behavior* assumption, not only for system calls (Guerra-Manzanares, Bahsi, & Nomm, 2019a; Guerra-Manzanares, Nomm, & Bahsi, 2019c) but also for run-time API calls (Alzaylaee, Yerima, & Sezer, 2017; Gong et al., 2020), thus suggesting that the nature and characteristics of the devices and versions used may actually matter. These previous studies highlighted the potential relevancy of the issue but did not analyze it thoroughly. This study fills this research gap by focusing on testing and analyzing the cross-device behavioral consistency issue. The main novelties provided by this work are an extensive benchmarking of the same set of apps in a large collection of devices and the exploration of the implications of the observed *inconsistencies* on ML-based detection systems that use dynamically acquired features.

To the best of our knowledge, this is the first study that explores the phenomenon thoroughly and demonstrates, using nine different Android devices and two different Android versions on the same data set, that cross-device consistency cannot be assumed and that not considering the heterogeneity of data sources that may coexist in production setups (e.g., users devices data, honeypots, sandboxes, and threat intelligence feeds) can introduce *measurement bias* (Data, 2021) and significantly erode the detection performance of the classifier.

The experimental setup of this research explores the behavioral differences in system calls invoked by the same set of applications in different types of Android platforms (i.e., nine devices) and OS versions (i.e., Android 9 and 10) and evaluates the implications of such differences for effective ML-based malware detection. Furthermore, the possible causes behind the observed differences are assessed and recommendations to minimize their impact on production setups are suggested.

This paper is structured as follows: Section 2 gives background knowledge, while Section 3 provides an overview of related literature. Section 4 details the methodology followed in this research. The main results are provided in Section 5. Section 6 addresses discussion points and limitations of this investigation, and Section 7 outlines the main takeaways.

2. Background information

Kernel calls or *system calls* (*syscalls*, for short) are an interface between a running process (i.e., application in *User* mode) and the operating system *kernel*, which controls the underlying hardware (Bovet & Cesati, 2005). More precisely, syscalls enable running programs to request services and access resources on the system (e.g., memory, storage, I/O, etc.). As all requests from applications pass through the system call interface before they are executed via hardware, the analysis of syscalls is a relevant source of information about the application's intentions (i.e., app behavior) (Malik & Khatter, 2016). Like any other running program in the system, malicious software must also use system calls to function and perform their desired actions on the system. Therefore, it is possible to *trace* and analyze system calls for discernible patterns that enable effective discrimination between benign and malicious behavior (Denney, Kaygusuz, & Zuluaga, 2018).

The usage of system call auditing as a means of malicious intrusion detection emerged long before the ubiquity of mobile devices and apps proliferation (Forrest, Hofmeyr, Somayaji, & Longstaff, 1996; Kosoresow & Hofmeyer, 1997) but has, since then, been widely adopted as an effective method for Android malware detection, thus making system calls the most used dynamic feature for such a purpose (Liu, Xu, Xu, Zhang, Sun, & Liu, 2020).

The collection of system calls requires the execution of the app in a *sandbox*, a controlled live environment. Both *real devices* and Android *emulators* have been widely used in research as collection platforms. A real device refers to an actual physical handset powered by an Android OS version, whereas an emulator is a software instance running on a *host* machine that simulates the dynamics and capabilities of a real device (Android, 2021d). In brief, the collection process involves the execution of the app in the selected environment and the log of the invoked syscalls at run-time. Some studies let the app run freely after booting it up (i.e., with no further interaction) for the whole collection time (Guerra-Manzanares et al., 2019a, 2019c), whereas others stimulate the app's interface by manually or automatically injecting events to trigger behaviors and expand *code coverage* (Tam, Feizollah, Anuar, Salleh, & Cavallaro, 2017). Both approaches are explored in this study.

3. Literature review

Static and *dynamic* features collected from Android applications are used to build Android malware detection systems (Liu et al., 2020). Static features are extracted from the app archive, source code, or metadata without the need to execute the application. Android *security permissions* and *API calls* are widely used static features for generating effective detection systems (Cai, Jiang, Gao, Li, & Yuan, 2021; Onwuzurike, Mariconti, Andriotis, Cristofaro, Ross, & Stringhini, 2019; Zhu, Li, Li, Li, You, & Song, 2021). Static features are relatively easy to acquire and provide extensive code coverage. However, they can easily avoid detection when code obfuscation techniques (e.g., encryption or polymorphic techniques) are used to hide the malicious code aiming

to bypass static features-based malware detection systems (Alzaylaee, Yerima, & Sezer, 2020a).

Dynamic features require executing the app in a live and controlled environment to be acquired. System calls are the most widely used dynamic features for Android malware detection (Liu et al., 2020). Even though they can be bypassed (Petsas, Voyatzis, Athanasopoulos, Polychronakis, & Ioannidis, 2014a), dynamic features-based detection methods are robust to code obfuscation and encryption techniques. However, they are more time-consuming and difficult to collect as they require the app to be installed and run in a sandbox device, either an emulator or a real device. Greater code coverage can be achieved when using pseudo-random events (Alzaylaee et al., 2020a).

This study focuses on the usage of system calls as detection features in the Android ecosystem, and, more specifically, on the evaluation of device influence on the collected behavior. In this regard, in the research literature on dynamic-featured Android detection systems, the selection of collection platforms is usually a discretionary decision mainly guided by the resources at hand and overall needs. As a result, no clear dominance of a *device* type is observed in the related literature. While some studies prefer the usage of real devices, using either single (Ahsan-Ul-Haque, Hossain, & Atiquzzaman, 2018; Amin, Zaman, Hossain, & Atiquzzaman, 2016; Arshad, Shah, Wahid, Mehmood, Song, & Yu, 2018; Canfora, Medvet, Mercaldo, & Visaggio, 2015; Da, Hongmei, & Xiangli, 2016; Isohara, Takemori, & Kubota, 2011; Saracino, Sgandurra, Dini, & Martinelli, 2018; Wahanggara & Prayudi, 2015; Xiao, Fu, Xiao, Jiang, Li, & Lu, 2015; Xiao, Xiao, Jiang, Liu, & Ye, 2016; Xiao, Zhang, Mercaldo, Hu, & Sangaiah, 2019) or multiple handsets (Alzaylaee, Yerima, & Sezer, 2020b; Burguera et al., 2011; Shabtai, Kanonov, Elovici, Glezer, & Weiss, 2012; Vidal et al., 2017; Yu, Zhang, Ge, & Hardy, 2013), others are inclined to emulators, using either specialized sandboxes for malware analysis (Feng, Ma, Sun, Xu, & Ma, 2018; Jang, Yun, Woo, & Kim, 2014; Lindorfer, Neugschwandtner, & Platzer, 2015; Naval, Laxmi, Rajarajan, Gaur, & Conti, 2015; Saif, El-Gokhy, & Sallam, 2018; Tam, Fattori, Khan, & Cavallaro, 2015; Tchakounte & Dayang, 2013; Yuan, Lu, Wang, & Xue, 2014) or *standard* Android emulators (Abderrahmane, Adnane, Yacine, & Khireddine, 2019; Afonso, de Amorim, Gregio, Junquera, & de Geus, 2015; Ananya, Aswathy, Amal, Swathy, Vinod, & Mohammad, 2020; Bernardi, Cimitile, Distante, Martinelli, & Mercaldo, 2019; Bhatia & Kaushal, 2017; Casolare, De Dominicis, Iadarola, Martinelli, Mercaldo, & Santone, 2021; Dimjasevic et al., 2016; Ferrante, Medvet, Mercaldo, Milosevic, & Visaggio, 2016; Guerra-Manzanares et al., 2019b; Hou et al., 2016; Jaiswal, Malik, & Jaafar, 2018; Kapratwar, Di Troia, & Stamp, 2017; Leeds, Keffeler, & Atkison, 2017; Malik & Khatter, 2016; Sihag, Vardhan, Singh, Choudhary, & Son, 2021; Singh & Hofmann, 2017; Surendran & Thomas, 2022; Surendran, Thomas, & Emmanuel, 2020a, 2020b; Tong & Yan, 2017; Vinod, Zemmari, & Conti, 2019). Even though most of the studies report the collection platform, they are vague in the description of the experimental setup and, more significantly, in the Android version used and device configuration. Furthermore, when different devices are used in the same experimental setup, usually the same OS version is used in all devices. As a result, the data gathered from a particular device and its specific configuration are generalized as representative of all Android systems. Therefore, the main assumption across all these studies either implicitly or explicitly (Burguera et al., 2011; Lin et al., 2013; Vidal et al., 2017), is that the behavior of the applications is fully consistent across Android platforms/devices and OS versions and that, consequently, these variables do not affect the collected behavior of the app.

However, the results of the small proportion of studies that experimented with both kinds of devices challenge the validity of this axiom (Alzaylaee et al., 2017; Guerra-Manzanares et al., 2019a; Guerra-Manzanares, Bahsi, & Nomm, 2021; Guerra-Manzanares et al., 2019c). Significant behavioral *inconsistencies* were found in dynamically collected data using distinct Android environments for API calls (Alzaylaee et al., 2017) and system calls (Guerra-Manzanares et al., 2019a, 2021, 2019c). Despite their observations, these studies did not perform a thorough inspection of the observed differences and their implications in machine learning-based malware detection systems. A remarkable research gap that is addressed in this research by collecting and analyzing the behavior of the same set of applications on a large set of Android platforms and OS versions, assessing their similarity, and evaluating the impact and implications of the behavioral differences in the effectiveness of ML-based malware detection models.

4. Methodology

This section outlines the methodology used in this research, which aims to assess the validity of the *cross-device consistent behavior* assumption. More precisely, the data collection process and the subsequent analysis performed are detailed in the following paragraphs.

4.1. Data set

Most of the research studies regarding ML-based Android malware detection use relatively large, labeled data sets (i.e., benign and malware samples) and a single Android device as a collection platform. In general, the larger the data set, the more tendency to use emulators is observed, as they are easier to deploy, manage and integrate into automated detection solutions (Dimjasevic et al., 2016). On the contrary, due to its different focus, this study uses a reduced data set and a large diversity of testing platforms. The fact that all the apps used in this research had to be successfully executed in a wide variety of hardware architectures (i.e., devices) and OS versions (i.e., *compatible* with all collection platforms) significantly limited the number of available data samples for our experimental setup. Consequently, this study does not aim for statistical significance but advocates for thorough data analysis and representativeness of the results. Thus, sixteen Android apps (i.e., eight malware and eight benign) belonging to different timeframes were tested on nine different devices (i.e., three real devices and six emulators) powered by two different versions of Android OS.

To provide a sound comparison of the extensive benchmarking and thorough data analysis performed in this research, the data samples used had to meet strict requirements, so the generation of the data set became a critical part of the whole methodology. The data set requirements are outlined as follows.

- The selected apps had to be successfully installed and executed on each testing platform. This aimed to ensure that any observed differences were not due to incompatibility or malfunctioning issues.
- Malicious and benign app sets had to include native libraries compiled for different hardware architectures (i.e., application binary interfaces or *ABI*s).
- Malicious samples had to represent widely spread families and both legacy and recent specimens (i.e., representative malware samples).

Due to the strict requirements of cross-device compatibility (i.e., architecture and OS version) and exploratory objective of this study, the data set aimed to be representative, priming quality over quantity. Further details about the data set are provided in the following paragraphs.

4.1.1. Malware set

The malware samples were selected from two different data sets that are publicly available for research purposes. Samples were randomly selected and included in the data set if they met all the requirements. Therefore, if in the testing phase, a randomly chosen sample did not meet the requirements (e.g. it was unable to be installed or executed on all the platforms), it was discarded and another application was randomly selected for testing. The malware samples represent different well-known Android malware families. Four samples of recent malware specimens were gathered from the *AndroidMalware 2020* repository

(Sk3ptre, 2020), and four *older* samples were acquired from the *CICAndMal2017* data set (Lashkari, Kadir, Taheri, & Ghorbani, 2018). Table 1 provides a detailed overview of the malware data set. It details the general malware category of the sample, the malware family it belongs to, MD5 hash, *VirusTotal*'s *first submission* date (i.e., used to date the sample) and the compatible architectures (i.e., ABIs supported). Additionally, for posterior reference of the apps, a unique identifier code was assigned to each sample (i.e., *MX*, where *X* is an integer between 1 and 8).

4.1.2. Benign set

The benign samples were gathered from the *CICAndMal2017* data set and were tested using VirusTotal's scan engine to confirm their non-malicious nature. Like the malware samples, the apps were selected randomly and included in the data set if they met the requirements. Their properties are described in Table 2. More precisely, app type (i.e., broad app category), MD5 hash, sample date (using the same data source as the malware applications), and the supported architectures are reported. Again, for posterior reference of the apps, a unique identifier code was assigned to each sample (i.e., *BX*, where *X* is an integer between 1 and 8).

4.2. Benchmarking platforms

A complete *testbed* of Android devices must include real and virtual devices, as they are both widely used for research purposes. Furthermore, to extensively analyze the possible differences in system calls on different Android platforms, different versions of the OS should be tested and controlled as *confounding* variables. Therefore, in this research, three real mobile handsets running two different Android OS versions (i.e., Android 9 and Android 10) were selected as benchmarking devices. To properly assess behavioral differences across devices, the same phone models were *virtualized* as accurately as possible using *Android Studio 4.1.2 Android Virtual Device (AVD) Manager* and *Genymotion Desktop 3.2.0* emulator. As system call data collection requires *superuser* privileges on the system, the real devices were *rooted* using *Magisk Manager* (Magisk, 2021), whereas the emulated devices had *root* available through system images. All the devices used 4G connection (i.e., one SIM card per device) and a 16 GB SD card.

4.2.1. Device specifications

The *host* platform for all the virtual devices was an Intel Core i7-8665U x86 64 CPU machine with 32 GB RAM running *Ubuntu 20.04 LTS*. A 4G USB modem was used for internet connection during the testing phase. Table 3 provides an overview of the parameters of the testing devices, both for real smartphones and their emulations, including details of their central processing unit (CPU), system on a chip (SoC) model, and *kernel* version. For posterior reference, a two-character unique identifier is assigned to each device. The first character identifies the nature of the device (i.e., R for real, A for Android Emulator, and G for Genymotion emulator), and the second character refers to the device number (i.e., an integer ranging from 1 to 3). Since R1 and R2 devices were running a 32-bit OS on a 64 bit CPU, system images with x86 architecture were chosen for E1 and E2 (i.e., their corresponding *Android emulator* virtualizations) to match the real device kernel architecture, while R3, running a 64-bit OS, was emulated as E3 with a x86 64 system image (i.e., currently *Android SDK* does not offer *ARM* system images for Android 9/10). Such distinctions were not possible in Genymotion-based instances (i.e., *GX* devices), as it only provides 32-bit x86 system images.

4.2.2. Device settings

The emulator instances were configured to resemble the real devices as accurately as possible using the options available in the emulator software. Consequently, if possible, all the emulated devices' specifications (e.g., OS image, CPU cores, RAM size, display resolutions and dpi, storage, SD card, etc.) were selected according to the real devices' specifications. However, some minor deviations remained since some features like storage size in Genymotion or display *dpi* sizes were available only in fixed configurations. In such cases, the nearest value to the real specification was used.

Regarding the user data configuration, all testing platforms were also given as similar settings as possible to mimic an *average* user setting. For instance, location services were enabled, WiFi connectivity was enabled (although only 4G connection was used), *Google Play Services* were enabled and the devices were logged in using a real Google account. However, in the case of Android Studio, the system images only permitted Google APIs but not Google Play Services, whereas Genymotion allowed using Google Play through the *Open GApps* widget. *Play Protect* was disabled on all platforms to grant unhampered installation and execution of malware applications.

4.3. Data collection

The purpose of the data collection phase was to *trace* and log the system call data for each executed application on each Android device under the same conditions for posterior analysis. For this task, *Android debug bridge* (*ADB*) (Android, 2021a) was used in combination with *Monkey* (Android, 2021e) and *Strace* (Strace, 2021) tools. ADB enabled the communication between the devices and the host machine via a command-line interface, while *Monkey* and *Strace* were used to inject pseudo-random events and log the app behavior (i.e., system calls), respectively.

Between test runs, the devices were cleaned up to ensure the same exact conditions for all samples. More precisely, after each malware data collection, the read-only memory (ROM) of the real devices was re-flashed with a clean system image, and, after each benign collection, a system factory reset was performed. After that, the real devices were re-rooted, and the OS settings configured again. This time-consuming and strict *restoration* policy ensured that the original system state was restored before each new collection and that all samples ran under the same exact conditions. In the case of emulators, for such a purpose, *snapshots* and *cloning* options were leveraged in Android Studio and Genymotion, respectively. More specifically, the sequential steps of the data collection procedure are detailed as follows:

1. Installation of the application
2. Application execution using the monkey tool and attachment of *strace* to the main app process
3. Running of the app for 5 min (i.e., freely or injecting pseudo-random events) and logging of the invoked system calls (i.e., behavior)
4. Detachment of *strace* from the main process, close of the application and *pull* of the log file to the host computer
5. Saving of the collected data and restoration of the clean system

As depicted in the diagram in Fig. 1, this procedure was performed twice per sample, for all samples in all devices, first with no user interaction (i.e., only execution, referenced as *1E*) and then simulating brief user interaction by injecting 50 pseudorandom events generated by the *monkey* tool (i.e., referenced as *50E*). In the exceptional cases when the app crashed during the collection process, the whole procedure was repeated according to the predefined criteria. No further interaction with the devices was performed at run-time. However, there was one forced deviation from this policy, which occurred before the execution of legacy applications in Android 10 devices. In these cases, a permissions-related screen prompt had to be accepted after installation due to a privacy-related addition on the Android 10 release (Android,

Table 1
Malware data set overview.

Code	Category	Family	MD5 Hash	Sample date	Supported ABIs
M1	Backdoor	*Mobok*	83763edd2d2e5d380df5c777cc9cdc24	2020-02-14	armeabi, armeabi-v7a, arm64-v8a, x86, x86 64
M2	Spyware	*XploitSpy*	117e1331306fec02b1ffe6b68d148cc9	2020-05-06	armeabi, armeabi-v7a, x86
M3	Adware	*Hiddad*	ec2b4ad861c0dbef1404713d9eac48a4	2020-03-13	all
M4	Spyware	*Bahamut*	9368dd657e410f8a9ba2b71c95cc0777	2020-08-26	all
M5	Adware	*MobiDash*	08d05f01671f788e9c17a9ffca0547b0	2016-02-09	armeabi, armeabi-v7a, arm64-v8a, x86, x86 64, mips, mips64
M6	Scareware	*Android.Spy.277*	2c5f158e2be5b0a67fe7378d6cff0d2d	2015-12-10	armeabi, armeabi-v7a, x86, mips
M7	Ransomware	*WannaLocker*	762138e933a681628ceab29d8e5a96a2	2017-07-25	all
M8	Fraudware	*Plankton*	0378f0cf4e7241a4c0f5a0722e601638	2013-08-07	all

Table 2
Benign data set overview.

Code	App type	MD5 Hash	Sample date	Supported ABIs
B1	Audiobook Player	ea30d7cc4c1dd7ad31bc32156fd2025b	2017-02-16	armeabi, armeabi-v7a, x86
B2	Timesheet organizer	0ea05f7634ac6b1003a774d3d7f22103	2016-09-07	armeabi, armeabi-v7a, x86, mips
B3	Graphic design tool	33b2fcb832c67a6c69a5cc05b0a44e3f	2017-02-07	armeabi, armeabi-v7a, arm64-v8a, x86, x86 64, mips, mips64
B4	PDF Converter	de76fdefa4a223d38162c8d349752720	2016-12-12	armeabi, armeabi-v7a, x86
B5	QR code scanner	90c81f6acc471d922fee136880eda641	2017-02-13	all
B6	Camping database	1607aef3d413ddd619c0248b07dd0087	2016-12-17	all
B7	Diary	944761948baeddf0e503325bf5e41ca4	2016-07-19	all
B8	Alarm clock	d93520ceee3ce2a3ff29a38cd7f6428c	2015-04-02	all

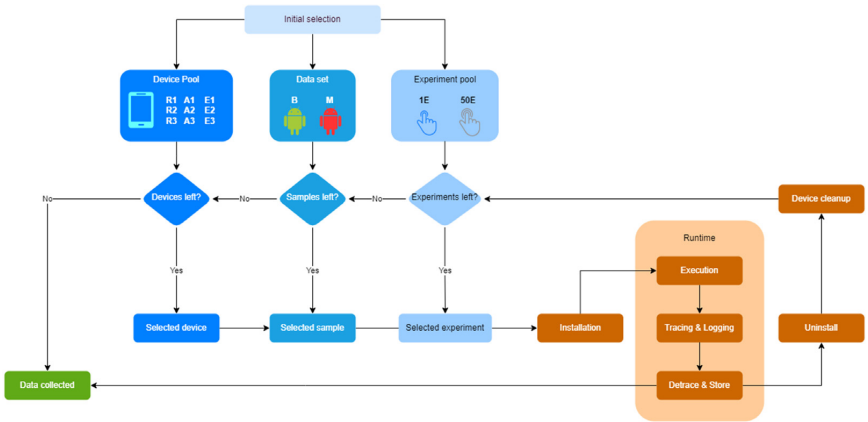


Fig. 1. Data collection workflow.

2021c). As a result, to collect data from legacy apps in Android 10, an acceptance button had to be pressed in the screen prompt with no more interaction needed. The *legacy* applications affected on Android 10 platforms by this feature were M5, M6, M7, M8, B2, B6, B7, and B8. The possible impact of these exceptional circumstances is discussed in Section 6.

As a result of the workflow depicted in Fig. 1, the outcome of the data collection process was a set of system call logs acquired for each malicious and benign application on each platform and both modes of execution: 1 event (execution-only) and 50 pseudo-random events. Therefore, as sixteen distinct samples were used in the experimental setup and run on two modes of execution possible on nine devices, a total of 288 logs were collected and analyzed.

4.4. Impact on machine learning-based detection

The data collected in the acquisition phase were processed to build and evaluate distinct machine learning-based classification models. More specifically, feature engineering was performed on the acquired data, and the absolute frequency (i.e., count) of each system call issued by the apps during the collection time was used to describe each application (i.e., data features). Thus, every data sample was described by a vector composed of syscall-based numeric features, which were used as input to the ML detection models. This data representation space provides a meaningful behavioral profile of the apps and has been used successfully to induce effective detection systems in previous research in the problem domain (Guerra-Manzanares et al., 2019b; Vidal et al., 2017). As a result, a tabular data set composed of 288 rows was generated, where each row (i.e., data sample) is described by the app code, device code (i.e., where the app was executed), mode of execution, class, and the system call frequency vector.

As machine learning models are sensitive to data quantity, the main aim of this experimentation was not the induction of effective forecasting models but the usage of machine learning models to assess the similarity among different acquisitions of the same application on different platforms and evaluate the implications of distinct behavioral profiles for the same application in simple detection models. For the sake of consistency, the same machine learning algorithm and hyper-parameters were used to induce all the ML-based detection models. The models' performance was evaluated using the *accuracy* performance metric, which informs about the degree of correctness of the model's predictions on the testing data. More specifically, it is calculated as the ratio of the number of correct class predictions over the total number of predictions (i.e., test samples). The accuracy metric is bounded in the [0, 1] range, and the higher the value (i.e., closer to one), the better the

Table 3
Specifications of the testbed devices.

Code	Device	System	CPU	Operating System
R1	Samsung Galaxy A20e	Model: SM-A202F RAM: 3 GB Storage: 32 GB	SoC: Exynos 7 Octa Instr. set: 64-bit ARMv8-A ABIs: arm64-v8a, armeabi-v7a, armeabi	Android: 9 (Pie) - API level 28 Kernel arch.: armv8l (32-bit) Kernel version: 4.4.111-...
R2	Samsung Galaxy A40	Model: SM-A405FN RAM: 4 GB Storage: 64 GB	SoC: Exynos 7 Octa Instr. set: 64-bit ARMv8-A ABIs: arm64-v8a, armeabi-v7a, armeabi	Android: 10 - API level 29 Kernel arch.: armv8l (32-bit) Kernel version: 4.4.177-...
R3	Xiaomi Redmi Note 8 Pro	Model: Note 8 Pro RAM: 6 GB Storage: 64 GB	SoC: MediaTek Helio G90T Instr. set: 64-bit ARMv8-A ABIs: arm64-v8a, armeabi-v7a, armeabi	Android: 10 - API level 29 Kernel arch.: aarch64 (64-bit) Kernel version: 4.14.141-...
A1	AVD Samsung Galaxy A20e	Model: AOSP on IA emulator RAM: 3 GB Storage: 32 GB	SoC: Android virtual processor Instr. set: 32-bit x86 ABIs: x86, armeabi-v7a, armeabi	Android: 9 (Pie) - API level 28 Kernel arch.: i686 (32-bit) Kernel version: 4.4.124+
A2	AVD Samsung Galaxy A40	Model: Android SDK built for x86 RAM: 4 GB Storage: 64 GB	SoC: Android virtual processor Instr. set: 32-bit x86 ABIs: x86	Android: 10 - API level 29 Kernel arch.: i686 (32-bit) Kernel version: 4.14.175
A3	AVD Xiaomi Redmi Note 8 Pro	Model: Android SDK built for x86 64 RAM: 6 GB Storage: 64 GB	SoC: Android virtual processor Instr. set: 64-bit x86 ABIs: x86 64, x86	Android: 10 - API level 29 Kernel arch.: x86 64 (64-bit) Kernel version: 4.14.175
G1	Genymotion Samsung Galaxy A20e	Model: Emulated A20e RAM: 3 GB Storage: 32 GB	SoC: i7-8665U (host) Instr. set: 32-bit x86 ABIs: x86	Android: 9 (Pie) - API level 28 Kernel arch.: i686 (32-bit) Kernel version: 4.4.157-...
G2	Genymotion Samsung Galaxy A40	Model: Emulated A40 RAM: 4 GB Storage: 32 GB	SoC: i7-8665U (host) Instr. set: 32-bit x86 ABIs: x86	Android: 10 - API level 29 Kernel arch.: i686 (32-bit) Kernel version: 4.4.157-...
G3	Genymotion Xiaomi Redmi Note 8 Pro	Model: Emulated Note 8 Pro RAM: 6 GB Storage: 32 GB	SoC: i7-8665U (host) Instr. set: 32-bit x86 ABIs: x86	Android: 10 - API level 29 Kernel arch.: i686 (32-bit) Kernel version: 4.4.157-...

classification performance of the model (i.e., more test data classified correctly).

In our case, the underlying idea behind the usage of these detection models is to leverage the model's *overfitting* capabilities to evaluate the similarities between the training and testing set. In general, an ML classifier model is said to overfit the training data when it is trained with limited data, and the trained model fits too closely the training data, thus not generalizing well to *unknown* or *new* data (IBM, 2021). This is an undesirable situation when building machine learning models that is usually addressed by providing more data or using regularization techniques. However, in our case, it is leveraged to provide a notion of the similarity between the training and testing sets, which are composed of exactly the same samples and described using the same set of features but reporting values collected in distinct Android platforms. Generally, a high-performance ML model should recognize almost perfectly training data when used as testing data. In our case, as the data set is small, it should be perfectly recognizable (i.e., 1 or 100% accuracy). Therefore, if an accuracy different from 1 is reported, then behavioral inconsistencies can be inferred as the only difference is the collection device, conditioning the behavioral profile registered (i.e., feature values). More precisely, the lower the prediction accuracy, the more dissimilar or inconsistent the behavior of the apps on the training device regarding the testing device.

5. Results

5.1. Data analysis

As a result of the extensive benchmarking performed, 288 log files were collected in the form of raw *system call trace logs*. The descriptive and statistical analyses of these data are provided in the following paragraphs.

The raw data logs evidenced differences, with varying proportions, in the length of collected sequences and the number of different system calls invoked by each app during the run-time. More significantly, this fact was observed for all data logs. Even though this observation was expected for distinct data samples, as different apps would show different behavior (i.e., invoke different number and set of syscalls), no notable deviation was expected for different executions of the same app in distinct devices for the same execution mode. However, the latter was not confirmed, and substantial differences were found in the data logs regarding the behavior of the same app in different devices for all combinations of apps and devices.

As the objective of this study was to explore app behavioral differences across different Android platforms and their implications for machine learning-based detection systems, the behavioral data were analyzed and compared between each real device and their emulated counterparts. Due to the extensive amount of data included in the logs, a sound comparison of syscalls at the individual level was deemed unfeasible, so the analysis focused on the total syscalls issued and the number of unique system calls invoked on the different devices. These data were extracted from the syscalls *trace* summaries. An example of a syscalls summary from *strace* logs is provided in Fig. 2. As can be observed, the summary data includes, among other data, the list of unique system calls issued (i.e., *syscall* column), the number of each unique syscall invoked (i.e., *calls* column), and the total number of all recorded system calls during the data collection process.

The total number of syscalls invoked at run-time and the number of unique system calls for each app run were used as primary data for the comparative analysis, as provided in Sections 5.2 and 5.3, whereas the frequency of each system call was used as input data to build and test the impact of the observed differences on ML-based classification models, as described in Section 5.4. The total figures representing the data recorded for each app on each platform are provided for 1 event in Table 4 and 50 events in Table 5. For each execution mode, the total number of invoked syscalls (i.e., *total* column) and the number of unique syscalls (i.e., *n* column) are provided for each combination of app and device. The device used is reflected in the columns, while the rows report the samples. Therefore, each row provides the behavior

Table 4
Extraction results summary (execution-only — 1E).

1 EVENT	REAL ANDROID PHONES						ANDROID SDK EMULATOR						GENYMOTION EMULATOR					
Sample	R1		R2		R3		A1		A2		A3		G1		G2		G3	
	Total	*n*	*Total*	*n*	*Total*	*n*	*Total*	*n*	*Total*	*n*	*Total*	*n*	*Total*	*n*	*Total*	*n*	*Total*	*n*
M1	1631	**13**	1819	**30**	3580	**38**	4434	**37**	8881	**36**	7674	**36**	3018	**36**	2385	**34**	2619	**34**
M2	7793	**50**	6604	**51**	15044	**50**	3463	**43**	5427	**46**	3485	**45**	2931	**47**	2235	**47**	2206	**46**
M3	3549	**19**	5916	**44**	16147	**46**	9208	**43**	1204022	**47**	12155	**43**	10693	**49**	10254	**47**	10361	**47**
M4	617	**27**	672	**26**	1094	**33**	2020	**28**	3280	**29**	1876	**25**	866	**29**	920	**28**	971	**28**
M5	6243	**30**	6825	**34**	6113	**30**	4539	**48**	846	**20**	313	**12**	12937	**53**	8646	**33**	8088	**32**
M6	594055	**36**	490404	**32**	4330	**19**	265683	**38**	581141	**30**	256816	**30**	186062	**45**	197392	**29**	171947	**28**
M7	17732	**38**	16900	**38**	17646	**34**	19297	**30**	34388	**36**	14457	**29**	14289	**12**	15845	**36**	13668	**33**
M8	4981	**28**	5049	**35**	6971	**26**	9459	**50**	49116	**26**	10238	**24**	7217	**52**	87740	**50**	249006	**26**
B1	2417	**32**	2912	**36**	9199	**38**	3569	**30**	7553	**32**	4664	**32**	1710	**36**	1710	**34**	1420	**34**
B2	627	**19**	1497	**25**	4278	**16**	4585	**49**	515	**17**	263	**16**	3742	**56**	3031	**56**	401	**18**
B3	553	**24**	4420	**49**	7575	**51**	6598	**48**	10889	**48**	6667	**45**	6357	**50**	4989	**48**	4892	**48**
B4	2287	**21**	206184	**54**	224182	**56**	23004	**52**	189200	**52**	9478	**49**	20044	**51**	6523	**50**	7424	**50**
B5	61	**8**	4602	**51**	7387	**52**	7650	**50**	13342	**51**	7150	**49**	6263	**52**	4676	**51**	4827	**51**
B6	2709	**24**	1886	**24**	1477	**24**	3325	**36**	5247	**24**	1740	**23**	3947	**40**	1096	**22**	1326	**22**
B7	2477	**37**	2096	**37**	2206	**37**	3815	**37**	4924	**24**	3642	**37**	2173	**40**	1745	**36**	1624	**36**
B8	147	**16**	1121	**27**	248	**17**	7382	**53**	402	**18**	237	**17**	3908	**55**	373	**21**	796	**24**

```
...
ioctl(44, BINDER_WRITE_READ, 0x7ff290f648) = 0
epoll_pwait(42,  <detached ...>
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 50.19    0.528000         771       684           epoll_pwait
 13.31    0.140000         286       489           ioctl
 12.17    0.128000         603       212        42 futex
  7.60    0.080000         197       405         1 read
  5.70    0.060000         163       367           write
  4.94    0.052000         151       344           getuid
  1.90    0.020000         106       187       131 recvfrom
  1.52    0.016000         122       131           sendto
  1.14    0.012000        1000        12           timerfd_settime
  0.76    0.008000         533        15           mmap
  0.38    0.004000        1333         3           clone
  0.38    0.004000         333        12           prctl
  0.00    0.000000           0         4           openat
  0.00    0.000000           0         5           close
  0.00    0.000000           0         1           writev
  0.00    0.000000           0         9           fstat
  0.00    0.000000           0         5           munmap
  0.00    0.000000           0        10           mprotect
  0.00    0.000000           0        27           madvise
------ ----------- ----------- --------- --------- ----------------
100.00    1.052000                  2922       174 total
```

Fig. 2. *Strace* summary log.

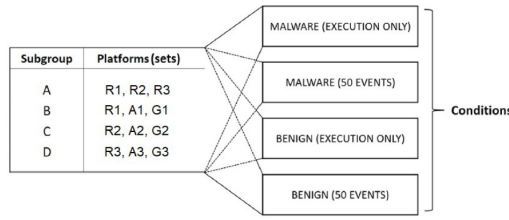


Fig. 3. Comparison groups and conditions.

of each sample on each device by specifying the total number of system calls issued (*total*) and the number of different system calls invoked (*n*).

The behavioral data provided in Tables 4 and 5 were analyzed for *consistency* and *similarity* across devices. In this regard, to establish a comprehensible analysis scope, the real devices were employed as the basis for four different device comparison subgroups. Such division provided distinctive sets of data for comparison. The generated subgroups were coded as A, B, C, and D for future reference. Subgroup A concentrates exclusively on behavioral differences among real devices, while subgroups B, C, and D assess the differences between each real device and their corresponding virtual devices. Furthermore, since data acquisition was performed on each device using two classes of apps and two modes of execution, it implied that there were four different perspectives (i.e., conditions) to examine the potential contrasts within each subgroup.

The four subgroups and the concept of *conditions* related to *class* and *execution mode* are presented in Fig. 3 and explored in the next section.

5.2. Comparison metrics

This section describes the data comparison performed via *scoring* metrics. In this regard, an experimental approach was implemented to adequately assess the primary differences or similarities within the data from the syscall summaries regarding the comparison sets and conditions. For the sake of comprehension, the results of subgroup A (i.e., comparison among real devices) under two of the four conditions (i.e., malware and benign samples with execution-only) are presented as descriptive examples (see Table 6).

It is worth mentioning that due to the presence of extreme values and several orders of magnitude within the collected data (i.e., the *total* values ranged from a few thousand to over a million syscalls), a numeric approach (i.e., *score*) was preferred to a graphical approach (i.e., bar or line charts) as the latter could hinder, distort or overemphasize the differences.

The four subgroups generated for comparison included data from three different platforms, compared under four different conditions using two numeric attributes (i.e., the *total* number of syscalls issued and the number of *unique* syscalls invoked).

Therefore, to perform a comprehensive comparison of both attributes, two distinct *similarity scores* were calculated to measure the differences of each data attribute across devices. They are described in the following paragraphs.

- For *total syscalls* data, the *ratio* of increase between two values was calculated (e.g., if the total syscalls data were 1500 and 1000 for the same app on two devices, the ratio would be reflected as 1.5, showing a 50% increase). All the calculations were performed pairwise, subtracting the smallest value from the largest. Thus the minimum possible value was 1, meaning that an equal number of *total* syscalls was invoked on both devices (i.e., no increase).
- For *number of unique* syscalls data, as they might show significant variability, the actual values (i.e., syscall name) instead of the summary figure were used. All the comparisons were performed pairwise, where the overlap between the unique syscalls sets (i.e., invoked on both executions) was used to calculate the *Jaccard coefficient*. The Jaccard coefficient (Costa, 2021) is a measure of similarity between sets calculated as the size of the intersection (i.e., overlap) over the size of the union of the sets, as expressed in Eq. (1). It ranges from 0 to 1, where the greater the value, the

Table 5
Extraction results summary (50 events injected — 50E).

50 EVENTS	REAL ANDROID PHONES						ANDROID SDK EMULATOR						GENYMOTION EMULATOR					
Sample	R1		R2		R3		A1		A2		A3		G1		G2		G3	
	Total	n	Total	n	Total	n	Total	n	Total	n	Total	n	Total	n	Total	n	Total	n
M1	4481	15	1910	30	5845	31	6766	31	10716	11	6626	35	5148	31	6199	30	2966	11
M2	8010	50	7002	51	6164	21	3635	46	6783	47	4145	46	3144	47	2523	47	2508	47
M3	2922	19	2334	22	5155	29	154294	27	11299	26	8283	26	7065	26	5666	26	3047	33
M4	576	26	603	26	615	26	1834	26	2497	26	1806	26	361	24	642	25	670	25
M5	6119	25	6188	30	5386	22	342	10	357	12	458	16	10161	30	8859	34	7225	24
M6	598843	35	502948	36	1437	12	259148	29	584115	30	128754	14	192390	29	210356	29	170547	28
M7	16988	38	16157	36	13642	36	16232	37	28584	34	14956	28	14041	11	13862	28	13305	12
M8	4543	29	4295	26	72619	22	4623	25	400113	26	44368	23	102560	29	76949	50	247626	27
B1	2622	32	26	5	449	13	3065	31	4470	31	3163	30	1041	30	998	31	279	12
B2	1107	19	171	15	44	12	392	18	1583	18	67	14	148	18	527	22	215	13
B3	2942	26	830	31	6828	32	7409	32	6309	28	255788	34	1694	30	2551	31	3992	33
B4	2146	20	1676	25	2889	18	2640	22	2503	21	1318	19	1921	21	1553	20	1390	20
B5	106	8	256	20	725	9	440	14	1790	27	708	16	496	20	392	19	641	15
B6	2524	24	1380	24	1468	24	2533	23	5378	24	1380	24	1713	22	808	22	1543	22
B7	2486	37	2060	37	874	25	3706	37	4907	24	3762	37	1890	37	1850	37	1490	36
B8	169	17	547	23	168	15	141	16	412	18	236	17	143	17	170	18	160	17

Table 6
Similarity comparison — subgroup A (execution-only).

	R1 vs. R2			R1 vs. R3			R2 vs. R3		
APK	Total sum	Uniques	Intersect.	Total sum	Uniques	Intersect.	Total sum	Uniques	Intersect.
M1	**1.12**	0.43	13(13;30)	2.19	0.34	13(13;38)	1.97	**0.79**	30(30;38)
M2	**1.18**	**0.94**	49(50;51)	1.93	**0.92**	48(50;50)	2.28	**0.98**	50(51;50)
M3	1.67	0.43	19(19;44)	4.55	0.41	19(19;46)	2.73	**0.96**	44(44;46)
M4	**1.09**	**0.96**	26(27;26)	1.77	**0.76**	26(27;33)	1.63	0.74	25(26;33)
M5	**1.09**	**0.83**	29(30;34)	**1.02**	**0.94**	29(30;30)	**1.12**	**0.88**	30(34;30)
M6	**1.21**	**0.84**	31(36;32)	137.20	0.53	19(36;19)	113.26	0.59	19(32;19)
M7	**1.05**	**1.00**	38(38;38)	**1.00**	0.71	30(38;34)	**1.04**	0.71	30(38;34)
M8	**1.01**	0.70	26(28;35)	1.40	**0.86**	25(28;26)	1.38	0.74	26(35;26)
	R1 vs. R2			**R1 vs. R3**			**R2 vs. R3**		
APK	Total sum	Uniques	Intersect.	Total sum	Uniques	Intersect.	Total sum	Uniques	Intersect.
B1	**1.20**	**0.89**	32(32;36)	3.81	**0.79**	31(32;38)	3.16	**0.85**	34(36;38)
B2	2.39	**0.76**	19(19;25)	6.82	**0.75**	15(19;16)	2.86	0.64	16(25;16)
B3	7.99	0.49	24(24;49)	13.70	0.47	24(24;51)	1.71	**0.92**	48(49;51)
B4	90.15	0.39	21(21;54)	98.02	0.38	21(21;56)	**1.09**	**0.96**	54(54;56)
B5	75.44	0.16	8(8;51)	121.10	0.15	8(8;52)	1.61	**0.98**	51(51;52)
B6	1.44	**1.00**	24(24;24)	1.83	**0.85**	22(24;24)	1.28	**0.85**	22(24;24)
B7	**1.18**	**1.00**	37(37;37)	**1.12**	**0.95**	36(37;37)	**1.05**	**0.95**	36(37;37)
B8	7.63	0.59	16(16;27)	1.69	0.74	14(16;17)	4.52	0.63	17(27;17)

Note: **Bold** – threshold reached; – overall similarity at least 0.75 (good); – overall similarity at least 0.90 (great)

more similarity between the sets.

$$J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \frac{|S_1 \cap S_2|}{|S_1| + |S_2| - |S_1 \cap S_2|} \quad (1)$$

The calculation of these scores resulted in two similarity indexes per pair of compared devices per sample, as can be observed in the example provided in Table 6, applied to execution-only data for subgroup A.

For the sake of interpretation of the results, *similarity thresholds* were established to qualify pairwise behaviors as *similar*. The similarity threshold was set to 0.75 for both comparative scores. Therefore, the behavior of a specific app on two different platforms was qualified as *similar* if the total syscalls ratio did not exceed 1.25 and the Jaccard coefficient did not fall below 0.75.

Table 6 provides both comparison metrics for subgroup A of data in execution-only mode. The *total sum* column provides the ratio for the *total syscalls* data between pairs of runs while the *Uniques* column provides the *Jaccard coefficient*. Besides, Table 6 also includes the size of the overlap of unique syscalls on the compared devices (i.e., *Intersect* column) and the total number of unique syscalls on both platforms reported within parentheses in the same column. *Similar* behaviors are highlighted in green. Besides, the pairs where the sets displayed an outstanding degree of similarity (i.e., the ratio of total syscalls below 1.10 and Jaccard coefficient greater than 0.90) are highlighted in dark green.

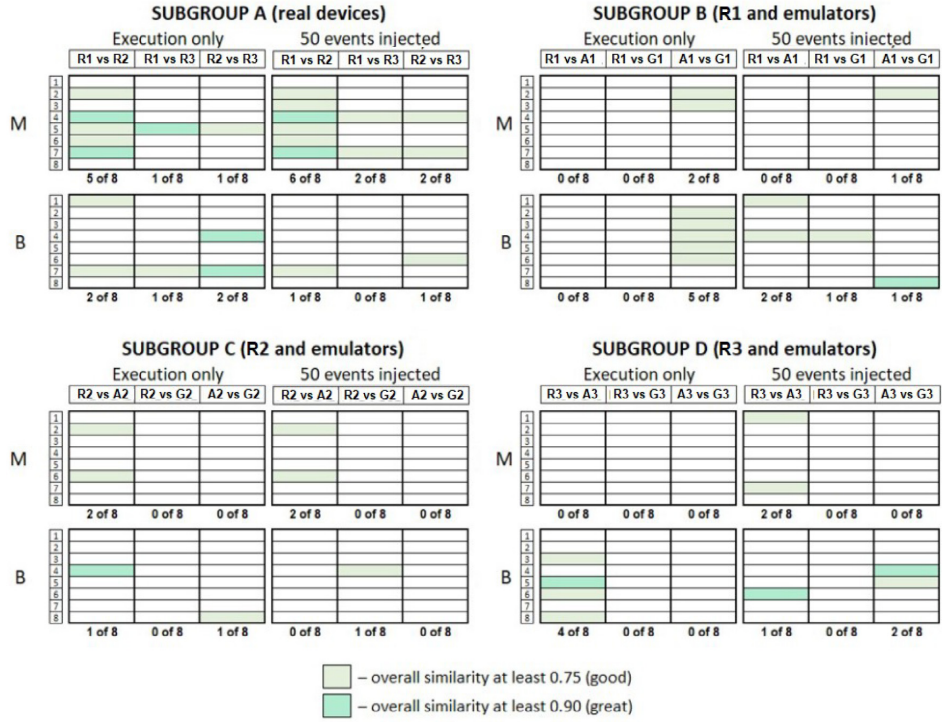
5.3. Cross-device behavioral analysis

The general overview of the results, covering the full spectrum of comparison sets and conditions, is displayed in Table 7. In this table, the same information as in Table 6 is provided but, for the sake of interpretation of the results, the numeric values have been omitted, thus providing a better visualization and comparison of the observed similarities. The same color patterns to highlight *similar* behaviors are applied. The following paragraphs summarize the results and provide the main findings.

5.3.1. App behavior: Real device comparison

The comparison of app behaviors on the real devices (i.e., subgroup A) evidences that, in general, the behavior of an app in distinct real devices is remarkably dissimilar as only 12 of the 48 compared behaviors in Table 6 are above the (*good*) *similarity threshold* (i.e., 0.75 for both attributes). From these, just 5 of them show very similar values with a *great* similarity score. Furthermore, the observed similarities seem to be present just under certain conditions. For instance, similar results are observed between R1 and R2 for malware samples in both modes of execution. However, these similarities are largely diminished if R1 and R2 behavioral profiles are compared with R3. In such a case, only a few malware samples continue to display consistency (e.g., M4 in the case of only-execution mode and M5 and M7 in the case of 50 events).

Table 7
General overview of comparison results.



Finally, for malware samples, slightly greater behavioral similarities across devices are observed when 50 events are injected.

In the case of benign samples, the opposed trends are observed. A decreased similarity is obtained when more events are injected and show, in general, greater dissimilarity than the malware samples in both execution modes. Therefore, for benign apps, there weren't notable general similarities observed among devices as they were in the malware case. a minor exception is spotted in the comparison between R2 and R3, where B4 and B7 achieved the 0.90 threshold for the execution-only condition. This execution mode shows also a noticeable similarity for the rest of apps when just the Jaccard coefficient is considered, meaning that the platforms invoked a similar set of unique system calls but not a similar amount of them.

Overall, as can be noticed in Table 7, this subgroup was the one showing more behavioral similarities among all the analyzed subgroups. However, despite the observation of some similarities, it can be concluded that the behavior of the applications was significantly dissimilar across different real devices and OS versions.

5.3.2. App behavior: Real devices compared to emulations

As displayed in Table 7, the comparatives among real devices and their emulated versions (i.e., subgroups B, C, and D) show remarkably inconsistent results. This fact evidences that the behavior of apps in real devices and their emulated versions, even when the virtual devices fully mimic the settings and properties of the real devices, are significantly different.

Although some exceptionally similar behaviors were spotted, no significant similarity patterns were observed in most of the analyzed sets under any condition. The only remarkable exception is observed in subgroup D, where R3 and A3 reached the *good* similarity threshold for four benign samples under the execution-only condition. Besides, it is worth noting that when these devices are considered, the eight benign apps show remarkable similarities when the set of unique system calls is compared, scoring over the specified similarity threshold. However, these similarities vanish under the condition of 50 injected events.

5.4. Impact on ML-based malware detection models

The absolute frequency (i.e., count) of each system call invoked by the apps during the collection time was used to describe each application and as input features to induce the ML-based models. The whole set of apps invoked a total of 81 different system calls during the data collection phase. As a result, the feature vector used to describe each app is composed of 81 syscall frequency values plus additional metadata features such as the app code, device code, mode of execution, and class.

The Random Forest algorithm (Breiman, 2001), which has been used in similar setups with outstanding performance (Guerra-Manzanares et al., 2019a, 2019b), was used to build all the classification models in this study. The default hyper-parameters of *scikit-learn* implementation were used (scikit learn, 2021). The *testing* accuracy score for all the induced models was retrieved. The obtained results are provided in Fig. 4 for execution-only data and in Fig. 5 for the 50 events data.

More specifically, the bar charts in Figs. 4 and 5 provide the results for cross-device detection accuracy for different training and testing data sets splits. The vertical axis in these figures provides the accuracy score, whereas the horizontal axis informs about the source of the testing data (i.e., the collection device). Therefore, each bar reports the testing set accuracy for each trained detection model. The color of the bars informs about the training data used to build the detection model (see legend). As can be observed, for each execution mode, nine detection models were induced, trained with distinct data from each device, and referenced with a different color in the charts. All the trained models were tested separately with the data collected on

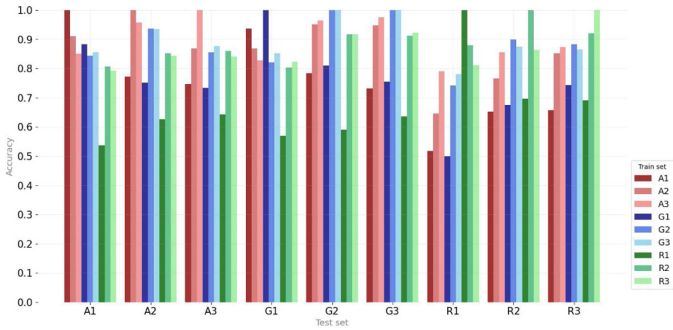


Fig. 4. Detection models accuracy (execution-only data).

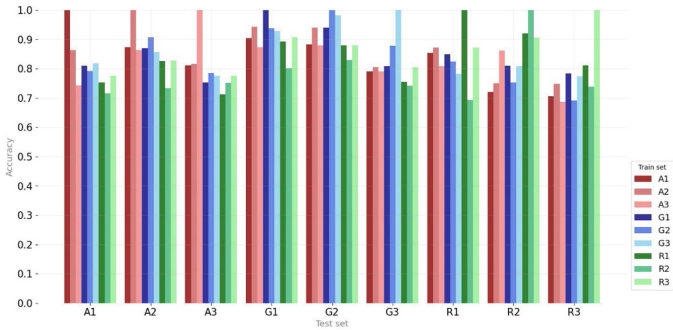


Fig. 5. Detection models accuracy (50 events).

all nine devices (i.e., including the training data) and *accuracy* was retrieved. It is worth remembering that the training and testing sets were composed of the same instances (i.e., the whole data set) in all cases. The only difference was in the feature values describing each sample which corresponded to the behavior collected on each particular device. The only exception occurred when the training and testing data belonged to the same device. In such a case, the training and testing data were identical (i.e., same feature values).

Given the reduced size of the data set and the inherent randomness of the classifier algorithm, to achieve a representative measure of the performance, each model testing was repeated 100 times. The *average* accuracy metrics are reported in Figs. 4 and 5.

The main idea behind these learning models is to analyze the impact of mixing distinct data collection sources in simple scenarios where the data set should be easily identifiable for the classifier model. In general, for any ML model, the greater the accuracy, the better the discriminatory capability of the model. In our case, the greater the accuracy, the more similar the behavior of the apps across devices and, consequently, the better the discriminatory capability of the model to cross-device data. Thus, if the behavior is fully consistent across different Android platforms, the average accuracy should be 1 (i.e., 100%) or very close, meaning that the model has no issues identifying class-related data collected from multiple devices. The lower the accuracy, the more dissimilar the behavior of the apps across devices and the worse the class-based discrimination by the model. Note that the discrimination of the training data (i.e., when used for testing) by the model should be close to perfect accuracy to make such implications, which means that the classification model must classify effectively the training data used to induce it.

The main difference between this approach and the previous data analysis is the class-level implications of the behavioral differences for ML-based detection classifiers, which aim to discriminate the class of a sample based on class-related patterns in the training data.

As can be seen in Figs. 4 and 5, the accuracy value is never reaching the maximum possible score (i.e., 1), except when the testing data are the same as the training data (i.e., the data used to build the model). This confirms the goodness of the induced models to perfectly discriminate their own data but not any other testing data, which corresponds to the same data set but described by the behavior collected on other devices. This indicates that, even though the samples on training and testing sets are the same, the data collection device impacts the performance of the induced systems as the class-level discrimination is significantly harmed even in this simple scenario. More accurately, except for G2 and G3 as training and testing data in only-execution mode (i.e., Fig. 4), all cross-device models decrease their performance significantly, showing that the behavior of apps in different devices is not consistent, thus confusing the classifiers induced with one specific device data to generalize effectively to test data collected in other Android platforms. Besides, in all cases, the induced model has no issues discriminating the data collected on the same device with perfect accuracy.

In summary, when all models are considered, the average cross-device accuracy for execution-only data is 0.80 with a standard deviation of 0.11, whereas for 50 events is 0.81 with a standard deviation of 0.07. Therefore, when more events are injected, the overall performance does not change significantly. However, the behavior appears to be slightly more consistent across devices, reflected by the reduced variability observed (i.e., smaller standard deviation). In conclusion, due to the behavioral differences of the applications across devices, the classifiers' cross-device data detection performance is notably reduced compared to the same-device data detection performance.

To further explore the implications of mixing behavioral profiles from different collection platforms, *mixed* models were induced using

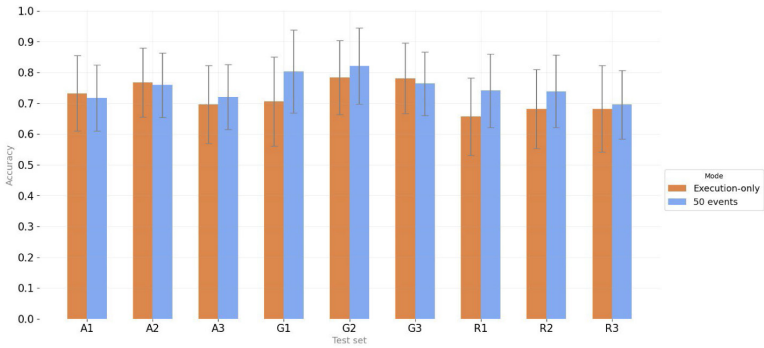


Fig. 6. *Mixed* models accuracy.

training sets constructed using random combinations of data from distinct devices. In this scenario, the benign and malware data were independently and randomly selected from the nine data sources, which ensured that the data set was composed of the sixteen samples. As in the previous case, the trained model was tested with the data collected from all devices. Thus the training sets included class samples from randomly selected devices, whereas the testing data belonged to a single device.

As all samples from a class were extracted from a single randomly selected device from the nine devices, in a minority of models (i.e., ≈ 11%) the same device was randomly assigned to provide both classes for the training data (as in the previous experiment). These models were kept for the sake of the completeness of the experiment and the representation of different production setups (i.e., data might or might not be mixed to build a model). For this experiment, a total of 10,000 models were induced. The average accuracy results are provided in Fig. 6 for both execution modes (i.e., orange for execution-only and blue for 50 events). The height of the bars reports the average accuracy value, while the standard deviation is depicted as extended gray lines below and above the average value. Given that the horizontal axis reports the testing data set, the accuracy score informs about the discriminatory properties of each data source in mixed trained models.

As shown in Fig. 6, the performance of the models where the training data is mixed is remarkably lower than on the models induced in Fig. 4 and Fig. 5, where the data for both classes were collected on the same device. More specifically, for the execution-only mode, an average accuracy of 0.72 is achieved with a standard deviation of 0.044, whereas for the 50 events mode an average accuracy of 0.75 is reported and a standard deviation of 0.038. More interestingly, data collected on the Genymotion emulator (i.e., *GX*) emerge as the most easily recognizable by the mixed models, while the data collected on real devices are the most challenging and provide the worst performance of all mixed models. Lastly, in general, the mixed models provide better performance for 50 events data than execution-only data. It may suggest that more events would tend to make the behaviors more similar across devices and, consequently, easier to discriminate by the mixed models.

In any case, these results evidence that cross-device consistent behavior cannot be assumed and that the data source must be considered in the design and data pipelines of any robust machine learning-based Android malware detection system. Furthermore, mixing different data sources in both training and testing sets seems to impact notably the classification models' performance. Our results show that, on average, in simple models and easy data sets, the cross-device accuracy of single-source trained models might be ≈ 20% lower than the same-device testing accuracy and ≈ 30% lower in the mixed models' case.

For the sake of reproducibility and further analysis, the generated data set and raw log files are made publicly available.[1]

[1] https://github.com/aleguma/android-testbed

6. Discussion

This section discusses the observed anomalies in the data, the possible causes of the cross-device behavioral differences, and the overall implications of the results. Threats to the validity of our results and future work are also discussed.

6.1. Outlier cases

As shown in Section 5.3, the similarities in system call summaries between different platforms were mostly exceptional, present only under certain conditions. The majority of comparisons indicated remarkable differences in the analyzed data attributes. As can be seen in Tables 4 and 5, the total number of syscalls for all apps varied significantly on the different testing platforms. This variability was in most cases in the same order of magnitude, thus considered *normal* dispersion. For instance, M1 issued between 1,631 and 8,881 syscalls in all the platforms. However, particular instances showed more extreme cases where the inconsistencies differed by several orders of magnitude. For instance, in Table 4, the data collection of M6 in all devices exceeded 150,000 syscalls except in R3, where it produced just 4,330 syscalls. The opposite case is observed for M3, as on A2 over a million syscalls were registered but did not issue more than 20,000 in any other device. In the context of the present study, such extreme differences can be considered as *outliers*, thus likely not representing the *normal* behavior of the app, generating the suspicion that underlying issues generated such extreme behavior. To address this issue, multiple collections of the specific instances were performed in the problematic devices. However, the same results were obtained in all iterations. Therefore, they were kept in the comparison table as they represent the real behavior of the samples in the devices. In any case, these *outlier* cases further emphasize the different cross-device behavior observed in this research.

6.2. Causes and implications

Despite the existence of *outliers*, the majority of the results did not display such extreme levels and can be considered inside a *normal* range. Still, the behavioral differences among the devices are evident for all samples. This section explores the possible causes behind the different behaviors and the implications of these differences for ML-based detection systems.

6.2.1. Real devices compared to their emulations

The comparison of the behavioral profiles between the real devices and their corresponding emulations proves that the use and distribution of system calls differ remarkably among devices. For instance, Genymotion emulations show more differences with the corresponding real

devices than Android Emulator, as only 2,08% (i.e., 2 of 96) of the total comparisons involving Genymotion devices lay over the similarity threshold (i.e., both originating from sample B4). In this regard, it is worth noticing that Genymotion only allowed the usage of 32-bit x86 system images, which might have seriously affected the comparison of G3 to its reference model, R3, which uses a 64-bit OS. Android SDK emulations fared slightly better with 14,58% (i.e., 14 of 96) of the behavioral profiles fitting into the 0.75 similarity threshold and even showing some consistency for benign applications, for example, when R3 is compared with its emulation A3.

The causes behind the differences are likely multifaceted, originating from both hardware and software-related aspects. All emulations in this study used *Intel* hardware architecture (see Table 3), whereas most of the real devices manufactured nowadays, including the phones used in this research, use *ARM* hardware architecture. In this regard, even though system calls in Linux are mostly universal for *user space* processes on different Linux devices, the number of system calls and their availability in distinct hardware architectures differ (Juszkiewicz, 2021). Besides, calling conventions are also distinct between x86 and ARM platforms (Kumar Jha, 2014a, 2014b). These primary differences are reflected in the logged *syscalls traces*. These differences in the usage and availability of syscalls imply that even though emulators are fast and convenient platforms for feature extraction and model training, the models induced using emulator-acquired data might not discriminate effectively test data collected from real Android devices. Besides, models built using real device data might also underperform when tested with data acquired on emulators. Both facts have been tested and verified in this study (see Section 5.4).

To address this issue, better *generalizable* models could be induced if an efficient ARM-based architecture is implemented on emulators. Thus real devices and emulators would share the same underlying architecture, and the differences based on this issue could be minimized. In this regard, even though they are not available nowadays, the recent updates indicate that Google might bring back ARM-based system images to the *SDK* emulator with the latest Android 12 version (Android, 2021b), which might enable new levels of architectural compatibility for emulator-based researchers.

Despite its importance, the architectural differences in system call handling may not be the only cause for the behavioral differences between real devices and their emulations. Modern malware can implement anti-emulation and sandbox detection techniques that may hide the malicious behavior in virtual environments (Petsas, Voyatzis, Athanasopoulos, Polychronakis, & Ioannidis, 2014b; Vidas & Christin, 2014). Therefore, some sophisticated malware might never reveal its malicious behavior in a virtual environment as emulators can mimic but never perfectly match real devices' capabilities and specifications. Based on our results, the collected profiles from malware instances do not appear to include such advanced techniques.

6.2.2. Comparison among real devices

This study employed three real Android devices. Although the comparison of system calls' summaries in this subgroup showed the most consistent results, as shown in Table 7, the similarities were present only under specific conditions. Most of the malware samples displayed relatively similar results on R1 and R2 devices for both modes of execution (i.e., more similarity in the case of 50 events). Such similarities vanish when R3 is included in the comparison. Additionally, the benign apps (except for B6 and B7) did not manifest fully consistent cross-device similarity, although unique calls data between R2 and R3 displayed good parity for execution-only mode.

Interestingly, the devices with the largest number of similarities, R1 and R2, were running different Android versions (9 and 10, respectively) but share the same kernel architecture (i.e., *armv8l*, a 32-bit version of *ARMv8*) and similar kernel series (i.e., 4.4.111 and 4.4.117, respectively). Furthermore, R1 and R2 are devices manufactured by the same Android OEM (i.e., Samsung devices). On the other side, R3 uses *Aarch64* architecture and a slightly newer version of the kernel. Despite using ARM architectures, R1 and R2 run 32-bit OS in a 64-bit chipset, whereas R3 implements a 64-bit OS. The significant differences in system call summaries among these devices indicate that, due to its improved instruction set, R3 implements different calling conventions for most apps.

In conclusion, different behavioral profiles are observed not only when real devices are compared to virtual devices but also among different real devices and OS versions, especially when they belong to distinct OEMs.

6.2.3. Implications for ML-based detection models

The detection models induced in Section 5.4 demonstrate that when training and testing data are collected on different devices, the performance of the detection system can decrease significantly as the detection system may fail to classify effectively new data samples collected on another device. Our experimental setup shows that this happens even when the features of the same set of samples used to induce the model are collected on a different device and used as testing data. Furthermore, based on our results, the cross-device performance of the induced models is further decreased when the training data are mixed, belonging to different collection devices.

The implications of these findings for production systems are significant. For instance, it implies that the detection models trained with data collected from emulators (in the cloud) may fail to recognize data collected from users' real devices (e.g., a local copy of the detection model is deployed for on-device detection or the data is sent to the cloud for detection). On the other side, given the myriad of different real devices available nowadays, a model trained with data collected from a single real device will not generalize well to data collected on other devices. Besides, as shown in Fig. 6, merging data from distinct data sources does not solve these issues and may provoke even more nefarious consequences in the detection performance. Therefore, the data source must be considered in any production ML-based malware detection system that integrates system calls as discriminatory features. Failing to attend to the behavioral differences collected on different devices might make the system fail by design.

The general recommendation based on our experimentation is to use data collected from a single device to train and test the models. This implies that emulators might be a better choice than using real devices as emulators are easier to control and deploy. Besides, on-device collection should be avoided as it may lead to ineffective detection when a local copy of the detection model is deployed on the devices.

In conclusion, due to the inconsistent behavior of applications across Android devices, *poor* cross-device detection performance should be expected if different data sources are mixed in production systems.

6.3. Threats to validity

The limitations of this study, which may threaten the validity of our results, are addressed in the following paragraphs.

- *Missing initial boot sequences*. Hooking the *strace* tool to a newly started app's main process through the ADB shell involves some delay. The length of this delay is variable and cannot be controlled. During that time, the initial system calls might not be logged, and, consequently, the variability of the *strace* delays might distort the final results gathered from different devices. Despite this limitation, *strace* was used in this research as it is the most used tool to acquire system calls in the related literature (Burguera et al., 2011; Guerra-Manzanares et al., 2019a; Vidal et al., 2017) and is usually included as a built-in tool in Android OS.
- *Unweighted unique system call values*. In our similarity analysis, the same weight was given to each unique system call invoked. Since the proportion of each system call was different on each device, the usage of a weighing or ranking system for the unique system calls could improve the similarity assessment.

- *Neglecting system calls sequences*. In our similarity analysis, we only considered the total number and set of system calls invoked by the applications on each device. However, it might be relevant to assess the degree of common sequences of system calls in the collected behaviors, that is, patterns of system calls among devices, as an additional metric for comparison and evaluation of the cross-device behavioral consistency.
- *Emulation detection*. Some of the malware used in this research might have been able to detect the virtual environment and might have not deployed their malicious activity. Although the malware apps were not reverse-engineered to examine anti-sandbox capabilities, according to the inspection of logs and results, these capabilities should not be assumed. However, even though the virtual devices mimicked the real device configurations, the possibility that some malware could have expressed a different behavior in the real devices from the virtual devices cannot be discarded. In any case, given that behavioral differences were also found for the benign samples, those results alone let us falsify the consistent cross-device behavioral assumption.

7. Conclusions

The primary objective of this research was to test the validity of the postulate that assumes consistent cross-platform behavior of Android apps as the basis for the generalization of the effectiveness of the proposed detection models in the related literature.

For that purpose, an extensive benchmarking setup composed of nine different devices was used. For *raw* data acquisition, strict criteria describing the setup settings, conditions, and restrictions were implemented. Samples of malicious and benign apps were installed and executed on testing environments according to predefined conditions (i.e., execution-only and 50 events) for extracting system calls-based behavioral data that were evaluated in a thorough comparative analysis. From the collected data, summary figures indicating the total number of system calls issued and the number of unique system calls invoked were extracted. The data were compared by groups according to devices and modes of execution. The similarity of the behavioral profiles of the apps across devices was analyzed using two similarity scores. Finally, the implications of the behavioral differences observed for the ML-based detection systems were evaluated.

The experimental results indicate the existence of important differences between real and virtual devices regarding system call usage and, consequently, the logged behavior of the apps. The differences among real devices were less salient but still significant. The evaluation of the impact of the differences on ML-based models showed a significant detrimental effect on the detection performance when the training and testing data are collected on different devices. Our empirical findings do not support the validity of the cross-device behavioral consistency of Android apps when system calls are used as descriptive features. Thus, if not carefully considered, data fusion may have destructive implications in the machine learning-based Android malware detection models.

CRediT authorship contribution statement

Alejandro Guerra-Manzanares: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Resources, Supervision, Visualization, Writing – original draft, Writing – review & editing. **Martin Välbe:** Conceptualization, Methodology, Software, Formal analysis, Investigation, Resources, Visualization, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Abderrahmane, A., Adnane, G., Yacine, C., & Khireddine, G. (2019). Android malware detection based on system calls analysis and cnn classification. In *2019 IEEE wireless communications and networking conference workshop (WCNCW)* (pp. 1–6). IEEE.

Afifi-Sabet, K. (2019). Millions hit by major android-based malware campaigns. https://www.itpro.co.uk/malware/33227/millions-hit-by-android-simbad-operation-sheep-campaigns.

Afonso, V. M., de Amorim, M. F., Gregio, A. R. A., Junquera, G. B., & de Geus, P. L. (2015). Identifying android malware using dynamically obtained features. *Journal of Computer Virology and Hacking Techniques*, *11*(1), 9–17.

Ahsan-Ul-Haque, A., Hossain, M. S., & Atiquzzaman, M. (2018). Sequencing system calls for effective malware detection in android. In *2018 IEEE global communications conference (GLOBECOM)* (pp. 1–7). IEEE.

Alzaylaee, M. K., Yerima, S. Y., & Sezer, S. (2017). Emulator vs real phone: Android malware detection using machine learning. In *Proceedings of the 3rd ACM on international workshop on security and privacy analytics* (pp. 65–72).

Alzaylaee, M. K., Yerima, S. Y., & Sezer, S. (2020a). Dldroid: Deep learning based android malware detection using real devices. *Computers Security*, *89*.

Alzaylaee, M. K., Yerima, S. Y., & Sezer, S. (2020b). Dldroid: Deep learning based android malware detection using real devices. *Computers & Security*, *89*, Article 101663.

Amin, M. R., Zaman, M., Hossain, M. S., & Atiquzzaman, M. (2016). Behavioral malware detection approaches for android. In *2016 IEEE international conference on communications (ICC)* (pp. 1–6).

Ananya, A., Aswathy, A., Amal, T., Swathy, P., Vinod, P., & Mohammad, S. (2020). Sysdroid: a dynamic ml-based android malware analyzer using system call traces. *Cluster Computing*, 1–20.

Android (2021a). Android debug bridge (adb). https://developer.android.com/studio/command-line/adb.

Android (2021b). Emulator release notes. https://developer.android.com/studio/releases/emulator.

Android (2021c). Privacy changes in android 10. https://developer.android.com/about/versions/10/privacy/changes.

Android (2021d). Run apps on the android emulator. https://developer.android.com/studio/run/emulator.

Android (2021e). Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey.

Arshad, S., Shah, M. A., Wahid, A., Mehmood, A., Song, H., & Yu, H. (2018). Samadroid: A novel 3-level hybrid malware detection model for android operating system. *IEEE Access*, *6*, 4321–4339.

AV-Test (2021). Development of android malware. https://www.av-test.org/en/statistics/malware/.

Bernardi, M. L., Cimitile, M., Distante, D., Martinelli, F., & Mercaldo, F. (2019). Dynamic malware detection and phylogeny analysis using process mining. *International Journal of Information Security*, *18*(3), 257–284.

Bhatia, T., & Kaushal, R. (2017). Malware detection in android based on dynamic analysis. In *2017 international conference on cyber security and protection of digital services (cyber security)* (pp. 1–6).

Bovet, D. P., & Cesati, M. (2005). *Understanding the linux kernel: from I/O ports to process management*. O'Reilly Media, Inc..

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32.

Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on security and privacy in smartphones and mobile devices* (pp. 15–26).

Cai, M., Jiang, Y., Gao, C., Li, H., & Yuan, W. (2021). Learning features from enhanced function call graphs for android malware detection. *Neurocomputing*, *423*, 301–307.

Canfora, G., Medvet, E., Mercaldo, F., & Visaggio, C. A. (2015). Detecting android malware using sequences of system calls. In *Proceedings of the 3rd international workshop on software development lifecycle for mobile* (pp. 13–20).

Casolare, R., De Dominicis, C., Iadarola, G., Martinelli, F., Mercaldo, F., & Santone, A. (2021). Dynamic mobile malware detection through system call-based image representation. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*.

Chebyshev, V. (2021a). It threat evolution in q1 2021. mobile statistics. https://securelist.com/it-threat-evolution-q1-2021-mobile-statistics/102547/.

Chebyshev, V. (2021b). It threat evolution in q2 2021. mobile statistics. https://securelist.com/it-threat-evolution-q2-2021-mobile-statistics/.

CheckPoint (2017). Copycat - an in-depth analysis of the copycat android malware campaign. https://www.checkpoint.com/downloads/resources/copycat-research-report.pdf.

Cimpanu, C. (2021). New grifthorse malware has infected more than 10 million android phones. https://therecord.media/new-grifthorse-malware-has-infected-more-than-10-million-android-phones/.

Cortes, C., Jackel, L. D., Chiang, W.-P., et al. (1995). Limits on learning machine accuracy imposed by data quality. *Vol. 95*, In *KDD* (pp. 57–62).

Costa, L. d. F. (2021). Further generalizations of the jaccard index. arXiv preprint arXiv:2110.09619.

Curry, D. (2021). Android statistics. https://www.businessofapps.com/data/android-statistics/.

Da, C., Hongmei, Z., & Xiangli, Z. (2016). Detection of android malware security on system calls. In *2016 IEEE advanced information management, communicates, electronic and automation control conference (IMCEC)* (pp. 974–978).

Data, A. (2021). Seven types of data bias in machine learning. https://www.telusinternational.com/articles/7-types-of-data-bias-in-machine-learning.

Denney, K., Kaygusuz, C., & Zuluaga, J. (2018). A survey of malware detection using system call tracing techniques.

Dimjasevic, M., Atzeni, S., Ugrina, I., & Rakamaric, Z. (2016). Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on international workshop on security and privacy analytics* (pp. 1–8).

Fedler, R., Schutte, J., & Kulicke, M. (2013). On the e ffectiveness of malware protection on android. *Fraunhofer AISEC*, *45*.

Feng, P., Ma, J., Sun, C., Xu, X., & Ma, Y. (2018). A novel dynamic android malware detection system with ensemble learning. *IEEE Access*, *6*, 30996–31011.

Ferrante, A., Medvet, E., Mercaldo, F., Milosevic, J., & Visaggio, C. A. (2016). Spotting the malicious moment: Characterizing malware behavior using dynamic features. In *2016 11th international conference on availability, reliability and security (ARES)* (pp. 372–381). IEEE.

Forrest, S., Hofmeyr, S. A., Somayaji, A., & Longstaff, T. A. (1996). A sense of self for unix processes. In *Proceedings 1996 IEEE symposium on security and privacy* (pp. 120–128). IEEE.

Gong, L., Li, Z., Qian, F., Zhang, Z., Chen, Q. A., Qian, Z., et al. (2020). Experiences of landing machine learning onto market-scale mobile malware detection. In *Proceedings of the fifteenth european conference on computer systems* (pp. 1–14).

Google (2021). Google play protect. https://developers.google.com/android/play-protect.

Guerra-Manzanares, A., Bahsi, H., & Nomm, S. (2019a). Differences in android behavior between real device and emulator: a malware detection perspective. In *2019 sixth international conference on internet of things: systems, management and security (IOTSMS)* (pp. 399–404).

Guerra-Manzanares, A., Bahsi, H., & Nomm, S. (2021). Kronodroid: time-based hybrid-featured dataset for effective android malware detection and characterization. *Computers & Security*, *110*, Article 102399.

Guerra-Manzanares, A., Nõmm, S., & Bahsi, H. (2019b). In-depth feature selection and ranking for automated detection of mobile malware. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy - ICISSP* (pp. 274–283).

Guerra-Manzanares, A., Nomm, S., & Bahsi, H. (2019c). Time-frame analysis of system calls behavior in machine learning-based mobile malware detection. In *2019 international conference on cyber security for emerging technologies (CSET)*. IEEE.

Hou, S., Saas, A., Chen, L., & Ye, Y. (2016). Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In *2016 IEEE/WIC/ACM international conference on web intelligence workshops (WIW)* (pp. 104–111).

IBM (2021). Overfitting. https://www.ibm.com/cloud/learn/overfitting.

Ilascu, I. (2020). Rainbowmix apps generate 150,000 in daily ad fraud profit. https://www.bleepingcomputer.com/news/security/rainbowmix-apps-generate-150-000-in-daily-ad-fraud-profit/.

Islam, Z. (2021). Android malware on the rise, google's os is more interesting" to cybercriminals than apple ios. https://www.techspot.com/news/91519-android-more-interesting-average-cybercriminal-than-ios-malware.html.

Isohara, T., Takemori, K., & Kubota, A. (2011). Kernel-based behavior analysis for android malware detection. In *2011 seventh international conference on computational intelligence and security* (pp. 1011–1015). IEEE.

Jaiswal, M., Malik, Y., & Jaafar, F. (2018). Android gaming malware detection using system call analysis. In *2018 6th international symposium on digital forensic and security (ISDFS)* (pp. 1–5).

Jang, J.-w., Yun, J., Woo, J., & Kim, H. K. (2014). Androprofiler: anti-malware system based on behavior profiling of mobile malware. In: *Proceedings of the 23rd international conference on world wide web* (pp. 737–738).

Juszkiewicz, M. (2021). Linux kernel system calls for all architectures. https://marcin.juszkiewicz.com.pl/download/tables/syscalls.html.

Kapratwar, A., Di Troia, F., & Stamp, M. (2017). Static and dynamic analysis of android malware. *In ICISSP*, 653–662.

Keizer, G. (2012). Massive android malware op may have infected 5 million users. https://www.computerworld.com/article/2500566/massive-android-malware-op-may-have-infected-5-million-users.html.

Kingsley-Hughes, A. (2021). The android apps on your phone each have 39 security vulnerabilities on average. https://www.zdnet.com/article/the-android-apps-on-your-phone-each-have-39-security-vulnerabilities-on-average/.

Kosoresow, A., & Hofmeyer, S. (1997). Intrusion detection via system call traces. *IEEE Software*, *14*(5), 35–42.

Kumar Jha, P. (2014a). How a system call is executed in x86 architecture?. https://learnlinuxconcepts.blogspot.com/2014/10/how-system-call-is-executed-in-x86.html.

Kumar Jha, P. (2014b). How is a system call executed in arm architecture?. https://learnlinuxconcepts.blogspot.com/2014/10/how-is-system-call-executed-in-arm.html.

Lashkari, A. H., Kadir, A. F. A., Taheri, L., & Ghorbani, A. A. (2018). Toward developing a systematic approach to generate benchmark android malware datasets and classification. In *2018 international carnahan conference on security technology (ICCST)* (pp. 1–7). IEEE.

Leeds, M., Keffeler, M., & Atkison, T. (2017). A comparison of features for android malware detection. In *Proceedings of the southeast conference, ACM SE '17* (pp. 63–68). New York, NY, USA: Association for Computing Machinery.

Lin, Y.-D., Lai, Y.-C., Chen, C.-H., & Tsai, H.-C. (2013). Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers & Security*, *39*, 340–350.

Lindorfer, M., Neugschwandtner, M., & Platzer, C. (2015). Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. *Vol. 2*, In *2015 IEEE 39th annual computer software and applications conference* (pp. 422–433). IEEE.

Liu, K., Xu, S., Xu, G., Zhang, M., Sun, D., & Liu, H. (2020). A review of android malware detection approaches based on machine learning. *IEEE Access*, *8*, 124579–124607.

Lysne, O. (2018). Static detection of malware. In *The huawei and snowden questions* (pp. 57–66). Springer.

Magisk (2021). Download magisk manager latest version 23.0 for android 2021. https://magiskmanager.com/.

Malik, S., & Khatter, K. (2016). System call analysis of android malware families. *Indian Journal of Science and Technology*, *9*(21).

McGowan, E. (2020). Another 21 malware apps found on google play. https://blog.avast.com/new-malware-apps-on-google-play-avast.

Microsoft (2020). Sophisticated new android malware marks the latest evolution of mobile ransomware. https://www.microsoft.com/security/blog/2020/10/08/sophisticated-new-android-malware-marks-the-latest-evolution-of-mobile-ransomware/.

Millar, S., McLaughlin, N., del Rincon, J. M., & Miller, P. (2021). Multi-view deep learning for zero-day android malware detection. *Journal of Information Security and Applications*, *58*, Article 102718.

Naval, S., Laxmi, V., Rajarajan, M., Gaur, M. S., & Conti, M. (2015). Employing program semantics for malware detection. *IEEE Transactions on Information Forensics and Security*, *10*(12), 2591–2604.

Networks, P. (2021). Why you need static analysis, dynamic analysis, and machine learning?. https://www.paloaltonetworks.com/cyberpedia/why-you-need-static-analysis-dynamic-analysis-machine-learning.

O'Dea, S. (2021). Mobile operating systems' market share worldwide from january 2012 to june 2021. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/.

Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E. D., Ross, G., & Stringhini, G. (2019). Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). *ACM Transactions on Privacy and Security (TOPS)*, *22*(2), 1–34.

Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., & Ioannidis, S. (2014a). Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the seventh european workshop on system security* (pp. 1–6).

Petsas, T., Voyatzis, G., Athanasopoulos, E., Polychronakis, M., & Ioannidis, S. (2014b). Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the seventh european workshop on system security* (pp. 1–6).

Saif, D., El-Gokhy, S., & Sallam, E. (2018). Deep belief networks-based framework for malware detection in android systems. *Alexandria Engineering Journal*, *57*(4), 4049–4057.

Saracino, A., Sgandurra, D., Dini, G., & Martinelli, F. (2018). Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, *15*(1), 83–97.

scikit learn (2021). Sklearn.ensemble.randomforestclassifier. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html.

Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., & Weiss, Y. (2012). "Andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, *38*(1), 161–190.

Sihag, V., Vardhan, M., Singh, P., Choudhary, G., & Son, S. (2021). De-lady: Deep learning based android malware detection using dynamic features. *Journal of Internet Services and Information Security (JISIS)*, *11*(2), 34–45.

Singh, L., & Hofmann, M. (2017). Dynamic behavior analysis of android applications for malware detection. In *2017 international conference on intelligent communication and computational techniques (ICCT)* (pp. 1–7).

Sk3ptre (2020). Androidmalware 2020. https://github.com/sk3ptre/AndroidMalware_2020.

Sophos (2017). Malware goes mobile: Timeline of mobile threats, 2004 – 2016. https://www.sophos.com/en-us/medialibrary/PDFs/marketing%20material/sophos-threat-infographic-ten-years-malware-mobile-devices.pdf.

Statista (2021). Number of detected malicious installation packages on mobile devices worldwide from 4th quarter 2015 to 2nd quarter 2021. https://www.statista.com/statistics/653680/volume-of-detected-mobile-malware-packages/.

Stokel-Walker, C. (2021). Google can take two months to remove malware apps from app store. https://www.newscientist.com/article/2286931-google-can-take-two-months-to-remove-malware-apps-from-app-store/.

Strace (2021). Strace - linux syscall tracer. https://strace.io/.

Surendran, R., & Thomas, T. (2022). Detection of malware applications from centrality measures of syscall graph. *Concurrency Computations: Practice and Experience*.

Surendran, R., Thomas, T., & Emmanuel, S. (2020a). Gsdroid: Graph signal based compact feature representation for android malware detection. *Expert Systems with Applications*, *159*, Article 113581.

Surendran, R., Thomas, T., & Emmanuel, S. (2020b). A tan based hybrid model for android malware detection. *Journal of Information Security and Applications*, *54*, Article 102483.

Tam, K., Fattori, A., Khan, S., & Cavallaro, L. (2015). Copperdroid: Automatic reconstruction of android malware behaviors. In NDSS symposium 2015 (pp. 1–15).

Tam, K., Feizollah, A., Anuar, N. B., Salleh, R., & Cavallaro, L. (2017). The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, *49*(4), 1–41.

Tchakounte, F., & Dayang, P. (2013). System calls analysis of malwares on android. *International Journal of Science and Technology*, *2*(9), 669–674.

Thinagarajan, B. (2021). Defense in depth: How samsung knox defeats mobile malware. https://insights.samsung.com/2021/09/20/defense-in-depth-how-samsung-knox-defeats-mobile-malware/.

Tong, F., & Yan, Z. (2017). A hybrid approach of mobile malware detection in android. *Journal of Parallel and Distributed Computing*, *103*, 22–31.

Vidal, J. M., Orozco, A. L. S., & Villalba, L. G. (2017). Malware detection in mobile devices by analyzing sequences of system calls. *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, *11*(5), 594–598.

Vidas, T., & Christin, N. (2014). Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM symposium on information, computer and communications security* (pp. 447–458).

Vinod, P., Zemmari, A., & Conti, M. (2019). A machine learning based approach to detect malicious android apps using discriminant system calls. *Future Generation Computer Systems*, *94*, 333–350.

Wahanggara, V., & Prayudi, Y. (2015). Malware detection through call system on android smartphone using vector machine method. In *2015 fourth international conference on cyber security, cyber warfare, and digital forensic (CyberSec)* (pp. 62–67). IEEE.

Whitwam, R. (2021). Android antivirus apps are useless — here's what to do instead. https://www.extremetech.com/computing/104827-android-antivirus-apps-are-useless-heres-what-to-do-instead.

Xiao, X., Fu, P., Xiao, X., Jiang, Y., Li, Q., & Lu, R. (2015). Two effective methods to detect mobile malware. *Vol. 1*, In *2015 4th international conference on computer science and network technology (ICCSNT)* (pp. 1041–1045). IEEE.

Xiao, X., Xiao, X., Jiang, Y., Liu, X., & Ye, R. (2016). Identifying android malware with system call co-occurrence matrices. *Transactions on Emerging Telecommunications Technologies*, *27*(5), 675–684.

Xiao, X., Zhang, S., Mercaldo, F., Hu, G., & Sangaiah, A. K. (2019). Android malware detection based on system call sequences and lstm. *Multimedia Tools and Applications*, *78*(4), 3979–3999.

Yaswant, A. (2021). New advanced android malware posing as system update. https://blog.zimperium.com/new-advanced-android-malware-posing-as-system-update/.

Yerima, S. Y., Sezer, S., & Muttik, I. (2014). Android malware detection using parallel machine learning classifiers. In *2014 eighth international conference on next generation mobile apps, services and technologies* (pp. 37–42). IEEE.

Yu, W., Zhang, H., Ge, L., & Hardy, R. (2013). On behaviorbased detection of malware on android platform. In *2013 IEEE global communications conference (GLOBECOM)* (pp. 814–819).

Yuan, Z., Lu, Y., Wang, Z., & Xue, Y. (2014). Droid-sec: deep learning in android malware detection. In *Proceedings of the 2014 ACM conference on SIGCOMM* (pp. 371–372).

Zhu, H., Li, Y., Li, R., Li, J., You, Z., & Song, H. (2021). Sedmdroid: An enhanced stacking ensemble framework for android malware detection. *IEEE Transactions on Network Science and Engineering*, *8*(2), 984–994.

Appendix 8

Publication VIII
A. Guerra-Manzanares, H. Bahsi, and M. Luckner. Leveraging the first line of defense: A study on the evolution and usage of android security permissions for enhanced android malware detection. *Journal of Computer Virology and Hacking Techniques*, *in press*, 2022

Journal of Computer Virology and Hacking Techniques
https://doi.org/10.1007/s11416-022-00432-3

ORIGINAL PAPER

Leveraging the first line of defense: a study on the evolution and usage of android security permissions for enhanced android malware detection

Alejandro Guerra-Manzanares[1] · Hayretdin Bahsi[1] · Marcin Luckner[2]

Received: 25 October 2021 / Accepted: 29 March 2022

Abstract
Android security permissions are built-in security features that constrain what an app can do and access on the system, that is, its privileges. *Permissions* have been widely used for Android malware detection, mostly in combination with other relevant app attributes. The *available* set of permissions is dynamic, refined in every new Android OS version release. The refinement process adds new permissions and deprecates others. These changes directly impact the type and prevalence of permissions requested by malware and legitimate applications over time. Furthermore, malware trends and benign apps' inherent evolution influence their requested permissions. Therefore, the usage of these features in machine learning-based malware detection systems is prone to *concept drift* issues. Despite that, no previous study related to permissions has taken into account concept drift. In this study, we demonstrate that when concept drift is addressed, permissions can generate long-lasting and effective malware detection systems. Furthermore, the discriminatory capabilities of distinct set of features are tested. We found that the initial set of permissions, defined in Android 1.0 (API level 1), are sufficient to build an effective detection model, providing an average 0.93 F1 score in data that spans seven years. In addition, we explored and characterized permissions evolution using local and global interpretation methods. In this regard, the varying importance of individual permissions for malware and benign software recognition tasks over time are analyzed.

Keywords Android · Permission · Machine learning · Malware detection · Concept drift · Mobile security

1 Introduction

Android malware is ubiquitous and deceptive [1]. Malicious applications disguise in many forms and shapes, constantly adapting in an ever-evolving *sophistication* trend since the early years of Android operating system (OS) [2,3], the leading mobile OS [4]. The open nature of Android and its massive spread make the popular OS an attractive target for cyber attackers, thus posing end-users at constant risk [5]. Furthermore, as mobile devices are increasingly becoming more integrated into our daily routine, from leisure time activities to work-related tasks, mobile security emerges as a central element for individuals and companies to prevent cyber attacks and protect the wealth of sensitive data that mobile devices manage and store [6].

Security permissions constitute the *first line* of defense against malicious threats in Android devices. *Permissions* are a Linux kernel-based Android OS built-in security feature that enables the system to control what apps can do and access, that is, their *privileges*. In Android devices, the user grants or denies access to apps to data and resources from the system via permissions, thus determining the apps capabilities on the system. The original Android OS permissions-based security model implemented an *accept all-or-nothing* policy upon installation. However, due to its critical importance as the first-line security barrier, the security model evolved after the release of Android 6.0 *Marshmallow* (i.e., API level 23) in 2015. Since then, the *new* permissions model lets the user decide to grant or deny specific permissions, related to *sensitive* data, for each app at run-time and not upon installation [7]. This improvement may have led to increased risk awareness, greater flexibility and situational control over the privileges of apps on the sys-

Alejandro Guerra-Manzanares
alejandro.guerra@taltech.ee

[1] Department of Software Science, Tallinn University of Technology, Tallinn, Estonia

[2] Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

tem, which in turn may have diminished the risk for some users, but it has certainly not eradicated the threat.

Furthermore, even though additional security mechanisms have been implemented at software [8] and hardware [9] level to overcome the traditional limitations of antivirus to detect malware in the mobile platform [10], malware authors have always found their way to bypass them [11]. Statistical figures show that the threat is not only alive but constantly evolving [12].

In recent years, machine learning (ML) techniques have been explored as a means to mitigate exposure to the threat, showing remarkable success even with evolved, *zero-day* and *obfuscated* malware samples [13]. ML-based malware detection systems use statistical properties of collected samples to *learn* about *known* data and make accurate predictions about *future* or *unknown* data (i.e., supervised learning). A wide variety of properties or *features* of malware samples have been used to build detection systems [14]. However, according to their nature, they are broadly categorized as *static* or *dynamic* features. Static features are directly extracted from the source code, without executing the app, whereas dynamic features require the execution of the app in a *live* environment to be acquired.

Permissions are the most used *static* features for Android malware detection purposes [14]. Despite the remarkable effectiveness reported by these detection systems, most of them are built on data belonging to *old* and short time frames within the whole Android history, thus neglecting the impact of time and malware evolution on data. For instance, *Drebin* [15], the most used data set in recent studies, provides data collected between 2010 and 2012. Therefore, this *old* data set becomes an *obsolete* representation of the actual threat landscape and it is not representative of the recent malware threats (e.g., the first ransomware for Android was discovered in 2013) [16].

In addition to that, most studies propose the usage of a reduced subset of features for optimal detection, obtained after applying feature selection methods to the *training* data used to build the detection model. These feature subsets may provide great discriminatory power on the training set timeframe but generate doubts about the generalization of such systems to future time frames, where significant changes may affect relevant data properties due to the natural evolution of malware and benign data, thus directly impacting the discriminatory power of features and, consequently, the detection performance of those systems [17]. As a result, these facts cast severe doubts about the long-lasting detection capabilities of systems built using old data to detect *recent* malware. Relevant features to detect malware effectively are prone to change as malware evolves, a phenomenon called *concept drift*. Neglecting concept drift has a potentially devastating impact on detection systems. Therefore, addressing the detrimental impact of concept drift becomes a critical issue to build effective and long-lasting machine learning-based malware detection systems.

This study aims to address these gaps by taking the impact of time into consideration when *permissions* are used as model features. Firstly, we modeled concept drift effectively by using a solution consisting of an ensemble of ML classifiers. Then, we applied *permutation feature importance*, a global *interpretation* method, to characterize the most relevant features per data period and understand the changes in the discriminatory power of permissions over time. Furthermore, a local interpretation method was used (i.e., Shapley values) to explain individual predictions and *locally* compare permissions evolution for specific malware families. As security permissions are inherently *interpretable* constructs, their characterization provides useful insights about Android malware intentions and its evolution. To the best of our knowledge, no previous study focusing on permissions performed any temporal characterization nor analysis of permissions' importance evolution.

A distinctive fact of our concept drift modeling and characterization is the comparative analysis of the impact of distinct *natural* feature subsets on the model's performance. These feature sets were formed considering the *natural* evolution of the permission set which is updated in almost every new API release. As a result, these feature sets were not selected using any feature selection method, thus not artificially generated nor optimized for a specific time frame.

Lastly, although permissions have been widely used in detection systems, they have been relegated to a *secondary* role, mainly used in conjunction with other static or dynamic features to enhance performance [18]. This study brings permissions back to the *primary* role by showing that using a reduced set of permissions and addressing concept drift, a long-lasting effective malware detection system can be built. This malware detection system showed consistent performance, averaging 0.93 F1 score, in a seven-year-long time frame.

The paper structure is as follows: Sect. 2 provides background information about Android permissions. Section 3 outlines the state of the art in Android malware detection systems using permissions. The methodology used in this research and main results are addressed in Sects. 4 and 5, respectively. Section 6 provides the main findings and highlights discussion points. Lastly, Sect. 7 describes the limitations of this work while Sect. 8 concludes the study and outlines future work.

2 Android security permissions

Everything in Android, from the contacts list to games, is an application and every application, for security reasons, runs in a restricted and isolated environment (i.e., sandbox).

SPI Journal: 11416 MS: 0432 TYPESET DISK LE CP Disp.:2022/8/22 Pages: 32 Layout: Large

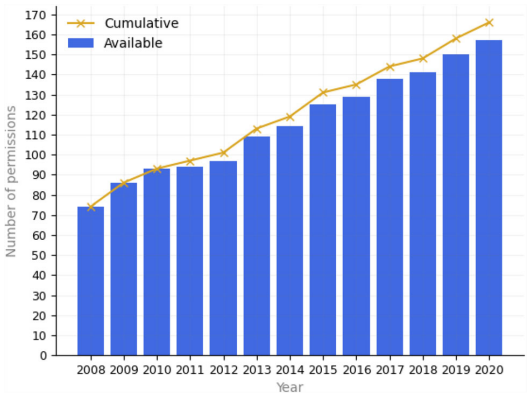


Fig. 1 Android permissions timeline evolution (based on data gathered from [21])

Therefore, if an app needs to access data or resources outside of its environment, it must ask for the necessary privileges to access them [19], requesting the appropriate permission.

In this regard, security permissions support user privacy by protecting access to *restricted data* (e.g., contacts information), preventing *restricted actions* (e.g., take pictures), and limiting interaction with other apps [20]. To be able to perform such actions or access sensitive data, the app must declare all the related permissions in the *AndroidManifest.xml*. The *manifest*, located in the root folder of the app's *.apk* archive, is the only mandatory file for every Android app and is used by the OS to get relevant information for the proper handling of the app.

The initial set of *standard* Android permissions available on the first Android release (i.e., API level 1) was composed of 74 permissions [21]. Since then, the set of available permissions has been modified, adding or deprecating permissions, on almost every new API release. This constant change has been in line with the increase in smartphone capabilities and the need for new security measures to handle them. The latest released version at the time of performing this research, Android 11 (i.e., API level 30), released in September 2020, has 157 available permissions. Thus, the permissions available suffered a two-fold increase in 13 years. More specifically, 166 permissions have been defined since API level 1 but 9 were deprecated in API updates. Figure 1 shows the evolution of Android permissions from 2008 to 2020, that is, from API level 1 (i.e., Android 1.0) to API level 30 (i.e., Android 11). The yellow line shows the cumulative number of permissions defined over time, while the blue bars indicate the number of *actually* available permissions for each API release/year (i.e., the usable set, without the deprecated permissions).

The dynamism of the phenomenon is evidenced in Fig. 1. The permissions set has increased more than two-fold since the first Android release and is constantly updated, especially in the most recent API releases. The updates have introduced new security permissions in response to new phone features (e.g., *USE_FINGERPRINT*, API level 23) or refine/extend existing permissions (e.g., *ACCESS_BACKGROUND_LOCATION*, API level 29, which refines *ACCESS_COARSE_LOCATION* and *ACCESS_FINE_LOCATION*) [21].

Besides, the usage of permissions is directly influenced by trends and behavioral changes in apps, affecting the prevalence of permissions over time. Therefore, the natural evolution of the feature set and the changes in prevalence over time make the usage of permissions for Android malware detection prone to concept drift issues.

Android security *standard* permissions are categorized into three risk or *protection* levels: *normal*, *signature*, and *dangerous* [21]. *Normal* permissions grant access to data and actions that pose a minimal risk to the user's privacy and the operation of other apps. *Signature* permissions are granted by the system to apps that declare a signature permission that another app has defined when both apps are signed with the same certificate. *Dangerous* permissions enable access to sensitive user data or actions that may affect the system and other apps. In addition to the *standard* permissions, developers can create their own permissions (i.e., *custom* permissions) which allow these apps to share their resources and data with other apps signed with the same certificate [22]. An additional category, outside of the scope of app developers are *special* permissions. These permissions are related to particular powerful app operations that only the platform and original equipment manufacturers can define [21].

Before Android 6.0 *Marshmallow*, all the permissions declared in the app manifest were granted automatically upon installation. So, if the user did not want to accept *all* the permissions the app declared, then the installation was not possible. This security paradigm changed in API level 23 with the inclusion of *run-time* permissions [23]. Since then, normal and signature permissions are granted automatically upon installation (i.e., *install-time* permissions), whereas dangerous permissions are requested for user acceptance during the app execution via an approval prompt (i.e., *run-time* permissions) [24]. Android 11 added further enhancements such as more granular permissions, *one-time* permissions, and the auto-reset of sensitive permissions for unused apps [25]. Recent Android releases have also focused on privacy issues and the update of the permission system [26], as evidenced by the refinement of the permission set and the addition of new permissions, as shown in Fig. 1. For instance, API level 29 introduced ten permissions and deprecated one, while API level 30 defined eight new permissions and removed one. These enhancements aim to provide more control to the users, transparency, and minimize data usage [20].

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

3 Related work

Android security permissions have been widely used in research since the early days of Android OS, becoming the most used static feature for Android malware detection [14]. Although static features are regarded as inherently *weak* against deception mechanisms such as encryption and obfuscation (i.e., especially API calls), Android permissions are relatively robust as without the required permission, the malicious behavior, obfuscated or not, might not be triggered [14]. Furthermore, permissions analysis enables direct detection on the device, upon download and without the need for app installation, execution [15] nor *root privileges* [27], featuring low computational cost and high efficiency [28]. Therefore, permissions constitute the first barrier to attackers, which could be leveraged for highly effective and efficient on-device malware detection.

Permissions have been widely used as input features to build Android malware detection solutions since the early days of Android OS. In this regard, they have been used alone, using all *available* permissions [27,29–31], selected subsets [17,32–34], or patterns and relationships between permissions [35–39], but also in combination with other static features extracted from the app manifest [18,40–42] and the decompiled source code [15,28,43–46]. Besides, so-called *hybrid* approaches have used permissions jointly with dynamic features such as system calls [47], run-time API calls/events [48,49], and network traffic [50,51].

However, despite the wide use of security permissions to build ML-based malware detection systems, no deep analysis of the feature evolution has ever been performed nor considered, thus neglecting the impact of the *time* variable on the learning models and its long-term evolution.

Among all the studies using permissions, only Hu et al. [46] considered concept drift. This study proposed a method to handle concept drift using a static feature set that includes permissions, API calls and actions (i.e., 405 features). Even though the method reports high accuracy, permissions are used in combination with other features, thus their analysis and impact on the overall performance of the detection model are not provided or explored. Besides, the data set used combines different data sets belonging to undefined and discontinued time frames (i.e., *Drebin* [15] and other unspecified sources), assuming the existence of sudden concept drift in the data. In this regard, a more *gradual* concept drift should be considered in a more realistic scenario where the threat landscape gradually evolves towards new threats based on old threats where sudden drifts might be possible but are less likely. In contrast, our study encompasses a long period in which a significant modification of the permissions set has been performed, thus capable of handling sudden and gradual drift, and our model only on permissions as input features, enabling us to assess and characterize using interpretability methods the impact of specific permissions and their evolution on the detection performance.

Besides, the vast majority of related research uses old, short, and static snapshots of Android malware historical data to build, validate and test their systems. For instance, *Drebin* [15] and *MalGenome* [52], the most used data sets for Android malware research, provide samples restricted just to the 2010–2012 time frame [53]. This fact poses serious concerns about the generalization capabilities of the proposed detection systems, built on old and non-representative data, to future and evolved malware.

Furthermore, in the studies that use data encompassing wider time frames to build ML-based detection systems (e.g., using two distinct data sets such as Drebin and Contagio [54]), the common practice is to merge the data and split the resulting data set randomly into disjoint sets of arbitrary proportions (i.e., the *training* and the *validation*/*testing* sets) [55]. This *typical* randomization procedure in machine learning, when applied to time-series data, introduces *temporal bias*, which has yielded not representative and overly inflated performances in Android malware detection research, not adjusted to the actual performance [56] due to the lack of *historical coherence* when data is split disrespecting the *historical timeline*. For proper validation, the testing data must belong to a *future* or *posterior* time frame regarding the training data [55,57]. An additional issue is that it is common that malware and benign samples used in the same data set do not belong to the same time frames, leading to an *historically* impossible configuration [44]. For instance, it is typical that benign data is collected at the time of research, whereas malware belongs to a well-known data set, such as Drebin, collected long before. This configuration generates biased detection systems and inflated performance as the features to describe apps belonging to distinct time frames might be too different (e.g., new permissions available, new requirements on permissions usage, etc.), thus generating an artificial scenario that does not reflect the real challenge of recognizing between malware and benign apps belonging to the same time frame.

These *common practices* provide unrealistic scenarios and neglect the impact of *time* on the input features and, consequently, its detrimental effect on the ML models over time (i.e.,*concept drift*). Therefore, neglecting concept drift provides biased and historically incoherent results, not adjusted to real scenarios [55–57].

Finally, the existing body of research has shown that the combination of permissions with other features may yield better performance than the usage of permissions alone [47,50,51,58], which has relegated permissions to a *secondary role*, as a complementary feature, mainly used in combination with other relevant features. Even though the combination may yield better detection performance, it does not provide any insights into the evolution of permissions,

SPI Journal: 11416 MS: 0432 TYPESET DISK LE CP Disp.:2022/8/22 Pages: 32 Layout: Large

their relevancy to the detection performance, or the changes in their discriminatory power over time.

This study focuses solely on the usage of permissions to detect Android malware, showing that when concept drift is addressed, permissions alone can provide and keep high-performance metrics over time. Notwithstanding that the combination with other features may yield increased performance, it is out of the scope of this research. To the best of our knowledge, no previous research has considered the concept drift issue for permissions, which has been explored for other features used in Android malware detection such as API calls [59,60] and system calls [61], nor performed a characterization of the phenomenon and assessed its impact on the detection performance over time.

4 Methodology

The following subsections detail the data set used in this research and concept drift basics, which enable the understanding of the approach used. After, the methodological workflow followed in this study, depicted in Fig. 2, is thoroughly explained.

4.1 Dataset and features

The data set used in this research is *KronoDroid* [53], a hybrid-featured data set that provides labeled samples of Android benign and malicious apps dating from 2008 to 2020, which makes it suitable for the analysis of the evolution of features and concept drift issues. Each sample in the data set is described by 489 features (i.e., 289 dynamic and 200 static). The dynamic features were collected using two distinct Android platforms, an emulator and a real device. Thus, the data set is originally split into two overlapping sub-datasets according to the source of the dynamic data. The data sets overlap in those samples that were able to provide dynamic data on both platforms but also contain distinct data samples, those acquired in only one of the sources. As in this research the interest lies just in the analysis of static features, disregarding the dynamic source, both data sets were merged to obtain the largest possible data set. The duplicated samples, caused by the overlap of the data sets, were removed. As a result, the data set used in this work is composed of 78,804 samples (i.e., 37,020 benign and 41,784 malware).

Regarding the input features, all the individual permissions-related features provided by the data set, the class label, and a timestamp were used. Therefore, each sample was described using 168 static features. More specifically, the permission-related features are composed of 166 categorical attributes, which are binary indicators of the presence of the permission in the app manifest (i.e., set as 1 if the permission is requested by the app and 0 if it is not). The data set provides permissions data until API level 30 (i.e., Android 11, released in September 2020). A total of 166 permissions were defined until API level 30 [21]. The timestamp used to locate each app within the Android historical timeline was the *last modification* timestamp. This timestamp is reported as the most reliable and accurate regarding the *historical context* from the four alternative timestamps provided by the data set. The last modification timestamp dates the app to the most recent timestamp retrieved from any of the app archive inner files [53]. Lastly, the class label (i.e., malware or benign) was obtained for each sample.

4.2 Concept drift

Static detection models assume that the statistical properties of the target distribution are relatively fixed, not changing over time. However, the malware threat landscape is dynamic, making the underlying distribution change over time (i.e., $P_t(\boldsymbol{X}) \neq P_{t+1}(\boldsymbol{X})$), referred to as *covariate shift*. When this shift affects the decision boundary of the classifier, and consequently, the class estimation (i.e., $P_t(\boldsymbol{Y}|\boldsymbol{X}) \neq P_{t+1}(\boldsymbol{Y}|\boldsymbol{X})$), the performance of the model is significantly harmed over time. Besides that, concept drift may also occur if the class estimation changes while $P_t(\boldsymbol{X})$ remains unchanged [62]. According to the speed of the changes, *incremental* drift, *gradual* drift, *sudden* drift, and *re-ocurring* drift have been proposed as concept drift typologies. The appearance of any of these typologies requires the update of the learning model [62]. Therefore, an effective malware detection model should be able to update its knowledge considering the ever-evolving threat landscape, keeping high detection performance in a non-stationary input context.

In this regard, to define concept drift for continuous analytics, let us define a new observation as $c_i = (x_i, y_i)$, where $x_i = (x_i^1, x_i^2, ..., x_i^n) \in \mathbf{X}$ is the feature vector and $y_i \in \mathbf{Y}$ is the target label. The incoming observations $c_i, \ldots, c_{i+k}$ are aggregated into sets of the same size k called *chunks*. To avoid the integration of extended periods in a single chunk, which may harm the detection of concept drift, temporal restrictions were imposed (i.e., a maximum of three months of data per chunk). In such a case, observation c_i is additionally described by a timestamp t_i and each chunk is a set $P_{[t_{min}, t_{max})} = \{c_i : t_{min} \leq t_i < t_{max}\}$, where t_{min}, t_{max} define the temporal borders of the given chunk.

Next, let us assume that features from two chunks are given by distributions F and F'. *Feature drift* is defined if the null hypothesis H_0 that F and F' are identical can be rejected [63]. Feature drift is categorized as concept drift if and only if the change in the distribution leads to a change in $\hat{y}$ estimation, which corresponds to a change in class estimation by the model in the described malware detection case. Therefore, concept drift, which relates to a change in the model's detection performance, is the focus of our analysis.

4.3 Workflow

The main objective of this work is to analyze the changes and evolution in permissions usage in Android apps and, more specifically, in malware (i.e., characterization). The focus lies on the impact of *time* on the detection capabilities of permission-based malware detection models (i.e., concept drift). Therefore, the optimization of classification quality is not a primary objective of this research. Nevertheless, for a meaningful analysis, the malware detector used during the analytic process must obtain high and preferably stable quality. The required stability of detection quality is challenged by the concept drift phenomenon observed in malware data [46,64], especially for long-term observations. The workflow performed in this research is provided in Fig. 2 and explained in the following subsections.

4.3.1 Data preprocessing

The feature vector defined for each app was composed of 166 permissions binary indicators, a timestamp, and the class label. As when some features were not available at the time, the vectors were filled with 0 values, this may have introduced noise in the learning models and provide biased results. To explore this issue, models with distinct permission set sizes were compared in this work. More specifically, the initial feature set from API level 1 (i.e., 74 permissions) and the whole feature set from API level 30 (i.e., 166 permissions) were evaluated. In the former case, the model was restricted to using the features from the initial set during the whole analysis period, whereas in the latter case, the model was free to select what feature sets provided better performance for each specific period.

However, before any analysis is performed, the feature set should be clear of redundant and irrelevant attributes that may affect the performance of the detection model and distort the characterization results. To address these issues, two sequential steps were performed. Firstly, sample variance was calculated for all features. The features with variance equal to zero were removed (i.e., constant or zero-valued) as they were irrelevant to building the discriminatory model. Next, highly correlated features were removed. The removal of strongly correlated features enabled us to eliminate redundant data as well as improve the quality of the data for the characterization step using the permutation importance technique [65]. To analyze the correlation between individual permissions, which are categorical features, *Kendall's rank correlation* [66] was used and calculated pairwise for all features. Last, as depicted in Fig. 2, in the final preprocessing step, data samples were divided into n sequential chunks defined by timestamps $t_1, \ldots, t_{n+1}$ as $P_{[t_1,t_2)}, P_{[t_2,t_3)}, \ldots, P_{[t_n,t_{n+1})}$.

4.3.2 Concept drift handling detection model

After the data preprocessing phase, the resulting data set was suitable to be processed by the malware detection model. The data set was analyzed chunk by chunk, simulating a realistic batch processing model for data streams [67]. Therefore, an algorithm that addresses the concept drift issue for stream data could be used to tackle the Android malware detection concept drift. In our solution, an existing method that uses a pool of classifiers was implemented [68]. More specifically, during the data stream processing, the algorithm dynamically selects the best ensemble of classifiers from an existing pool of classifiers to forecast the labels of the samples of the new data chunk. In our implementation, all the classifiers in the pool were random forest models, which yielded high performance in related studies [47,61,69,70]. Each classifier of the pool of classifiers is trained on a different historical/previous data chunk. This enables the pool to contain varied models and to expand its knowledge with data belonging to different time windows. This previous knowledge is used to predict the new data chunk. An updating mechanism is implemented by the algorithm every time a new data chunk is processed, introducing a new classifier trained on the newest chunk and removing the worst classifier (i.e., *aging* model), which causes the pool to be constantly modified after each chunk. The update mechanism aims to handle and address concept drift in the data. The original algorithm [68] was modified to address Android data particularities as described in [61]. More precisely, based on the implementation proposed by Guerra-Manzanares et al. [61], the following changes were performed:

1. The algorithm started with a full pool of classifiers and kept the pool size static during the whole analytic process. This change minimizes the differences in the quality of the detection process in its initial phases.
2. The pool of binary classifiers was supported by an anomaly detection model trained only with benign data. This modification improves the recognition of benign software, which was underrepresented in most data chunks and showed more consistent features over time (i.e., less *drift* in the data).

Malware detection can be defined as a binary classification problem, where *TP* (i.e., true positive) refers to the number of correctly recognized malware among all test instances. *TN* (i.e., true negative) reflects the number of correctly recognized benign software among all test data. *FP* (i.e., false positive) provides the number of actual benign samples incorrectly recognized as malware among all test samples and *FN* (i.e., false negative) the number of actual malware samples incorrectly identified as benign data by the classifier in the test set.

SPI Journal: 11416 MS: 0432 TYPESET DISK LE CP Disp.:2022/8/22 Pages: 32 Layout: Large

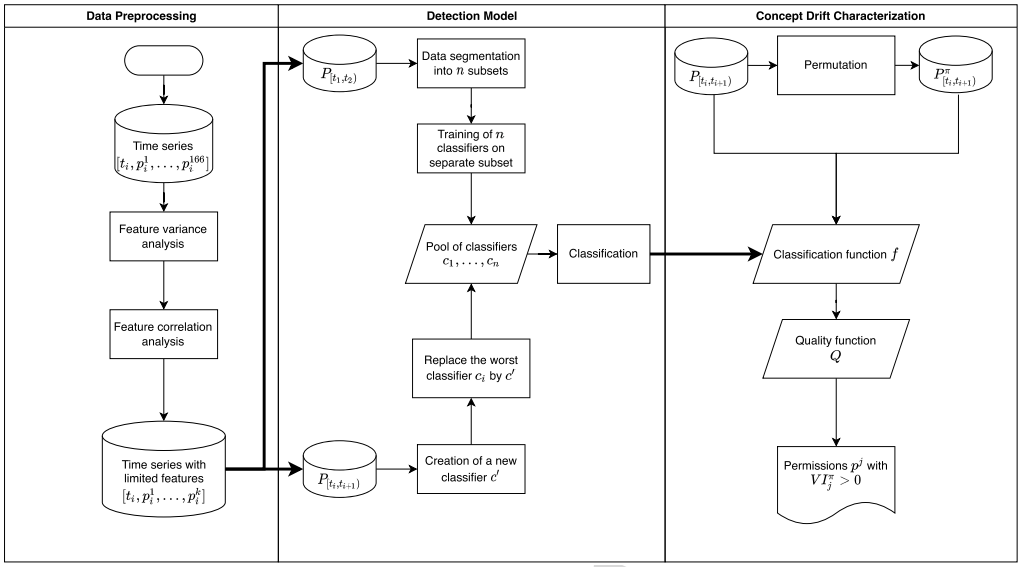


Fig. 2 Workflow of this study

Based on these concepts, the *F1 score* is used as a comprehensive metric for malware detection performance on imbalanced data sets. F1 score can be formulated as

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{1}$$

In our experimental setup, the F1 score metric was used to analyze $\hat{y}$ estimation performance of the permissions-based malware detection model for each chunk separately.

4.3.3 Data analysis and characterization

The F1 score gives a global picture of the detection performance of the proposed model. However, for a deeper analysis of permissions as a malware detection source, it is relevant to evaluate the influence of each permission separately on the detection process. As the detection model built was based on an ensemble of Random Forest classifiers, the built-in variable importance estimation [71] could be used for this task. In such a case, the *importance* calculation is tied to the optimization functions, which are generally *accuracy* or *F1* measures. However, for a better understanding of the analyzed data, it is more relevant to observe the influence of specific permissions on more specialized functions such as *specificity* and *recall*.

Specificity or True Negative Rate (TNR) is a measure of benign software acceptance (i.e., negative label recognition) and is calculated as:

$$TNR = \frac{TN}{TN + FP} \tag{2}$$

Recall or True Positive Rate (TRP) is a measure of the quality of malware detection (i.e., positive label recognition), defined as:

$$TPR = \frac{TP}{TP + FN}. \tag{3}$$

A preferred approach for analyzing features' importance using these specialized metrics is *permutation feature importance* analysis [72]. Permutation feature importance is a model-agnostic variable importance evaluation technique that is not tied to any particular metric or data set, suitable for analyzing the importance of model features for different quality metrics on the training and testing data sets. In this study, it was used as a tool to analyze the feature importance changes among distinct data chunks. More specifically, the permutation feature importance method evaluates how the random permutation of a feature influences the quality function calculated for the given decision model. For a matrix of feature values $\mathbf{X}$ with rows $\mathbf{x}_i$, given each of N observations and corresponding response y_i, $\mathbf{x}_i^{\pi,j}$ is a vector achieved by randomly permuting the jth column of $\mathbf{X}$, where π refers to a random permutation. According to [73], the permutation process should be repeated at least 50 times to obtain stable results. Originally, the method is defined for a loss function

SPI Journal: 11416 MS: 0432 TYPESET DISK LE CP Disp.:2022/8/22 Pages: 32 Layout: Large

L, but it can be easily adapted for the quality function Q with range $[0, 1]$ using $L(.) = 1 - Q(.)$. Then, the importance VI_j of the jth feature is calculated as the difference between the value of L for pseudo-randomized values and the value of L for the original data.

The proposed analytic tool uses the classification function $f_{[t_l, t_{l+1})}$ induced on data from chunk $P_{[t_l, t_{l+1})}$, where t_l and t_{l+1} define the temporal boundaries of a time period. The analysis observations X are taken from the corresponding chunk, $P_{[t_l, t_{l+1})}$. The procedure is summarized by the following equation:

$$VI_j^{\pi}(P_{[t_l,t_{l+1})}) = \frac{1}{N} \sum_{\substack{i=1, \\ x_i \in P_{[t_l,t_{l+1})}}}^{N} Q(y_i, x_i) - Q\left(y_i, f_{[t_l,t_{l+1})}(\mathbf{x}_i^{\pi,j})\right) \quad (4)$$

To analyze the importance of individual permissions for the recognition tasks, all features with $VI_j^{\pi}(.) > 0$ (i.e., positive importance) were retrieved for the analyzed data chunk.

Based on Eq. 4, the importance of features on the training set for the generated classifier is retrieved. The analysis of the important features for each new classifier's training set enables us to determine the relevant set of features for each specific time frame. When performed on consecutive data chunks, it allows the observer to analyze the changes in permissions' importance over time.

The permutation feature importance technique is a *global interpretability method* as it reports the average behavior or expected value of an ML model. Therefore, global interpretability methods are particularly useful to understand the general mechanisms in the data [74], which, in our case, enables us to study concept drift behavior. However, this wide-scope analysis may fail to grasp the particularities behind individual decisions. In such a case, *local interpretability* methods might provide a better answer. Local interpretability methods allow explaining the reasoning behind individual predictions. In our research, for the analysis of specific model predictions, *Shapley values* were used. Shapley values enable us to fairly attribute the prediction output among the relevant features, and it is the only local explanation method with a solid theoretical foundation [75]. The theoretical background of Shapley values lies in *coalitional game theory* [76]. The method aims to assess the contribution or importance of each feature in particular decisions (i.e., each feature is a *player* in a cooperative game or coalition where the prediction is the payout) [75]. As reflected in Eq. 5, the Shapley value of a feature (ϕ_i) is its contribution to the payout, weighted and summed over all possible feature combinations. It is calculated as:

Table 1 Data preprocessing results

Variance analysis	26 zero-valued
Correlation analysis	18 high-correlated
Final feature set	122 permissions

$$\phi_i(v) = \frac{1}{|N|} \sum_{S \subseteq N \setminus \{i\}} \binom{|N|-1}{|S|}^{-1} (v(S \cup \{i\}) - v(S)), \quad (5)$$

where N is the number of features, S is a subset of these features, i is the vector of feature values of the instance to be explained, and the function v calculates the payout for any subset/combination of features.

5 Results

The main results of the workflow followed in this research are described in the following subsections.

5.1 Data preprocessing

After the application of each sequential preprocessing step, the results reported in Table 1 were obtained.

From the initial set of 166 individual permissions, variance analysis showed that 26 were constant or null-valued for all data samples, so they were removed from the feature set as they did not provide any relevant information. Furthermore, correlation analysis reported that 18 features were highly correlated with at least another feature (i.e., Kendall's $|\tau| > 0.80$), evidencing redundancy in the data. These 18 features were removed. As a result, the final permissions feature set was composed of 122 permissions. This feature set is further referenced in this work as the *extended* or *full* feature set.

To test permissions evolution and explore the bias caused by the zero-filled values for the unavailable permissions, a reduced feature set was formed using only the permissions defined for API level 1 that remained available until API level 30. After the preprocessing steps, this *reduced* feature set was composed of 60 permissions.

Therefore, two feature sets were used in this research. The *extended* or *full* feature set includes all permissions defined during the whole Android history until API level 30 (i.e., 122), and the *reduced* or *initial* feature set is composed of the permissions defined in API level 1 that remained available until API level 30 (i.e., 60).

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

5.2 Detection model

The dynamic model described in Section 4.3.2 was used to classify and analyze malware and benign app samples described by permissions. The input data was described using both feature sets. The data samples were aggregated for each quarter of the 2011-2018 period (i.e., three months data chunks). This temporal constraint was found experimentally as the optimal for concept drift handling using these data. F1 score was calculated for each period using a dynamic ensemble selection from a pool of n classifiers as detailed in Section 4.3.2. In our experiment, $n = 12$ provided the best results.

Figure 3 illustrates the changes in the F1 score quality measure for malware detection among periods. The solid blue line shows the F1 score per chunk using the extended feature set, whereas the dashed blue line provides the same information for the reduced feature set. The horizontal yellow lines provide the average values for both cases.

As can be observed in Fig. 3, the performance of the proposed method to deal with Android data concept drift is relatively stable. Except for some periods (i.e., 5 chunks out of 28), the obtained results exceed 0.91 F1 score in all periods. The results obtained for the extended feature set and the reduced set are very similar, with a visible advantage of the reduced vector in the first period and a slight improvement of the full vector in the last periods. Furthermore, F1 score mean values are almost identical (i.e., ~ 0.93).

To formally compare both feature sets of input data, the non-parametric *Kolmogorov–Smirnov* test was used to analyze the equality of both probability distributions (i.e., *two-sample K–S test*). The results of the statistical analysis, depicted in Fig. 15 and further described in Appendix A, confirmed that there is no significant difference in detection performance between the compared feature sets.

A deeper inspection of Fig. 3 enables us to observe that, for both feature vectors, there are three visible quality drops in performance: in the initial period (i.e., 2011-Q3), 2015-Q4 and 2016-Q3. The nature of these drops is deeply analyzed and discussed in Section 5.3. In any case, despite these incidental dips, the system provided an average 0.93 F1 score in the analyzed period, which spans seven years of historical Android data.

In conclusion, based on the experimental results and the statistical analysis performed, depicted in Figs. 3 and 15, respectively, the discriminatory power of the *initial* feature set (i.e., 60 permissions) is comparable to the extended feature set (i.e., 122 permissions). This fact allows concluding that the set of initial permissions has a more significant role in malware detection than the later added permissions in the analyzed period. More features did not provide a better performance, emphasizing the goodness of a *small* subset of features to provide consistent and high malware detection performance over time, even in the presence of *evolving* data. Therefore, when concept drift is addressed, a high-performance malware detection solution can be built using just a small number of permissions as input features without needing to constantly update the feature set when new permissions are defined. Even though the importance of features changed over time (see Sect. 5.3), the initial feature set was found of critical relevancy to generate a long-lasting and robust permissions-based Android malware detection system.

5.3 Data analysis and characterization

The following subsections provide a characterization of the concept drift observed using interpretation methods. More precisely, Section 5.3.1. analyzes the evolution of the importance of individual permissions over time for recall and specificity tasks. Section 5.3.2. matches the class-based performance with the characterization analysis, and Section 5.3.3. performs a thorough analysis of the malware families' distribution over time and its impact on the learning models and concept drift.

5.3.1 Permissions' importance evolution

This subsection presents a thorough analysis of permissions' importance evolution over time. Features' importance was calculated using *specificity* and *recall* as quality (Q) functions. In this regard, importance calculated for specificity informs about which permissions are essential to recognize benign/legitimate applications. In the case of recall, it reports permissions important for the malware detection task. Permutation feature importance was calculated for the full and reduced feature sets separately for each quarter. For the sake of the stability of the results, the permutation feature importance procedure was performed four times with 2000 permutations per iteration.

Figures 4 and 6 provide the set of important features and their relative importance values calculated for the full feature set using specificity and recall, respectively. More specifically, each bar corresponds to a specific period (i.e., year quarter) where the color refers to a specific feature, as indicated by the graph legend. The bar range goes from 0 to 1, meaning that it contains the total feature importance for the period. The colored areas provide the relative feature importance of each permission regarding the total feature importance for that period. The vertical length of the areas is equated to the percentage or proportion of the total importance provided by each specific feature. For instance, in 2016-Q1 in Fig. 4, five colored areas are observed which relate to five different permissions found important, in varying proportions, to recognize benign apps in that specific period. Namely, SEND_SMS,

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

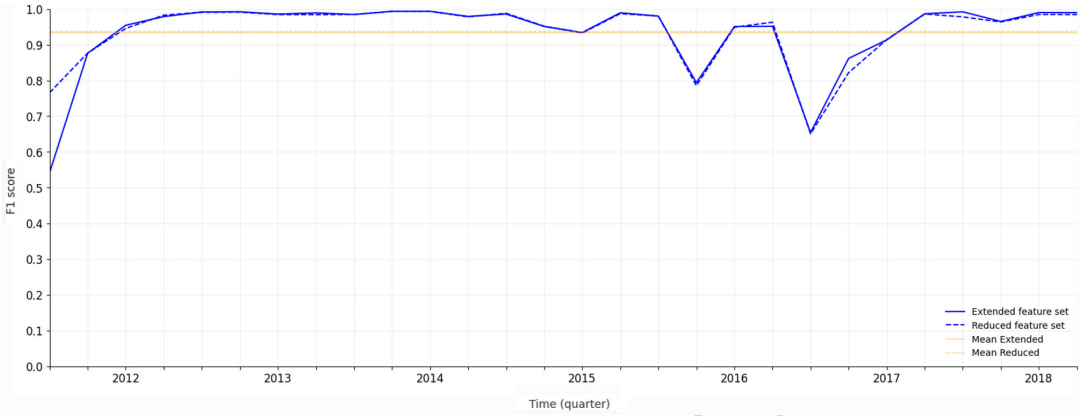


Fig. 3 Evolution of the detection performance using different feature sets

INTERNET, READ_PHONE_STATS, GET_TASKS and MOUNT_UNMOUNT_FILESYSTEMS permissions are represented by the purple, light green, dark blue, dark green and light blue areas, respectively. The grey regions refer to the importance related to features not present in the reduced feature set as, for the sake of interpretability and comparison between feature sets performance, in both Fig. 4 and 6, only the features belonging to the reduced feature set are depicted with colors and the ones belonging just to the extended feature set are reported in grey.

The white line provides the value of the maximum importance found for a feature in each specific period. For instance, in 2016-Q1 in Fig. 4, the most important feature, SEND_SMS, reports average feature importance of 0.35 (i.e., the value of the white line for that bar), meaning that when this feature was randomly shuffled, the specificity metric dropped, on average, 0.35 or 35%. Therefore, combining the information of the colored bars (i.e., relative importance) with the white line values (i.e., maximum absolute importance) provides a better depiction of the important features and their impact on the specific recognition tasks.

Figures 5 and 7 provide the same illustration of changes in feature importance for the reduced permission set. As can be noticed, no grey areas are found on these graphs. For the sake of comparison and interpretation of the results, in Figs. 4, 6, 5 and 7 characterization graphs, only the same set of permissions is depicted and with the same colors (i.e., only features belonging to the reduced set). Overall, as can be observed, the contribution of the later added permissions to the total importance (i.e., grey areas in Figs. 4 and 6) is less significant than the reduced permissions set. This observation aligns with the results reported in Figs. 3 and 15.

In the case of the extended feature set, 85 out of 122 features obtained average positive importance in at least one quarter, whereas for the reduced feature set, this value is 48 out of 60. Of these 48 features, 44 are present in both feature sets. The larger number of features important for the extended set causes a reduction of the importance value of particular features, as evidenced by the grey areas found in the extended set graphs (i.e., Figs. 4 and 6). However, even though the presence of these features does not allow the bars to be completely color-filled (i.e., except for 2015-Q3 in Fig. 6), the importance of the initial set of features is remarkably more significant in all periods, as they are responsible for more than 85% of the total importance of the whole time frame analyzed (i.e., 2011–2018).

For the specificity task, the distribution of the essential features is similar for both feature vectors, as depicted in Figs. 4 and 5. The most important permission in a single quarter is MOUNT_UNMOUNT_FILESYSTEMS (i.e., 2017-Q2). The four permissions with the largest overall importance (i.e., SEND_SMS, READ_PHONE_STATE, MOUNT_UNMOUNT_FILESYSTEMS, and INTERNET) are the same for both feature vectors. These results might be expected as they are permissions related to common mobile phone functions such as communications, phone calls, and access operations to storage file systems. The contribution to the importance of the subset of permissions only belonging to the extended permissions set is visibly lower than the importance of the permissions included in the reduced feature set. Even though the impact of the extended features has fluctuated over time, the common subset of features explains most of the feature importance in the whole analyzed time frame, thus being critical for effective benign software detection over time. Furthermore, the absolute importance values (i.e., white line) reflect that the importance of the most important feature per quarter is extremely high for this recognition task, averaging 26.9% for the extended feature set and 33.1% for the reduced feature set.

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

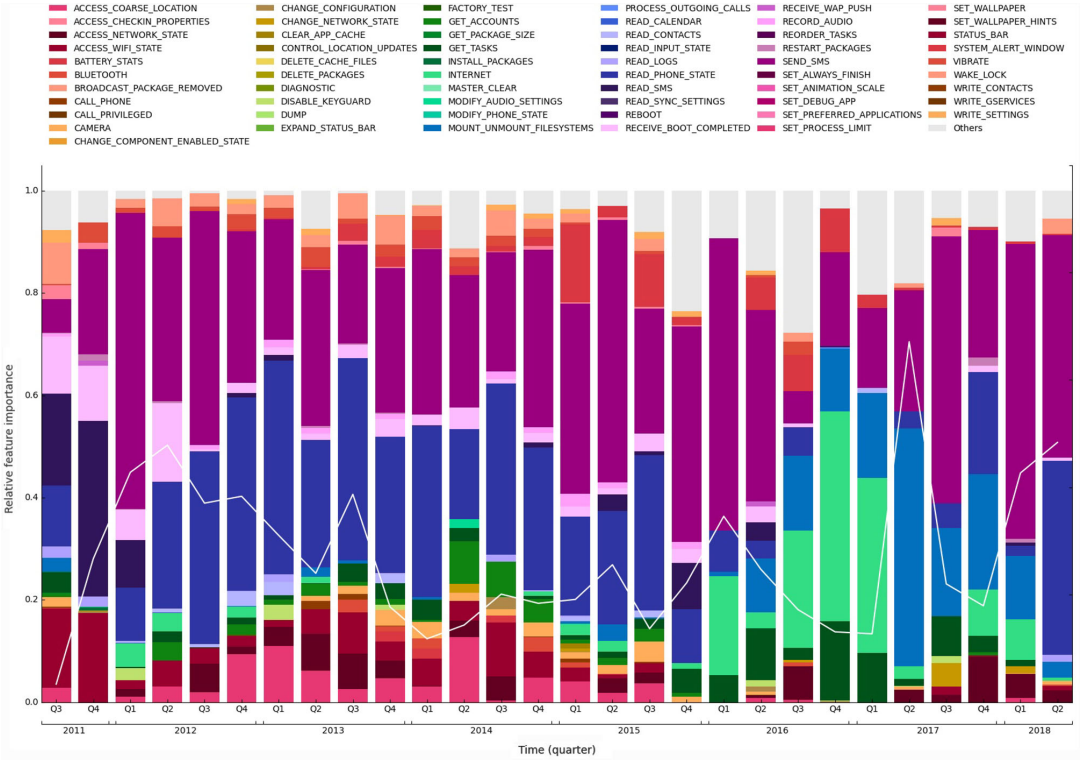


Fig. 4 Importance for specificity, calculated for the extended feature set

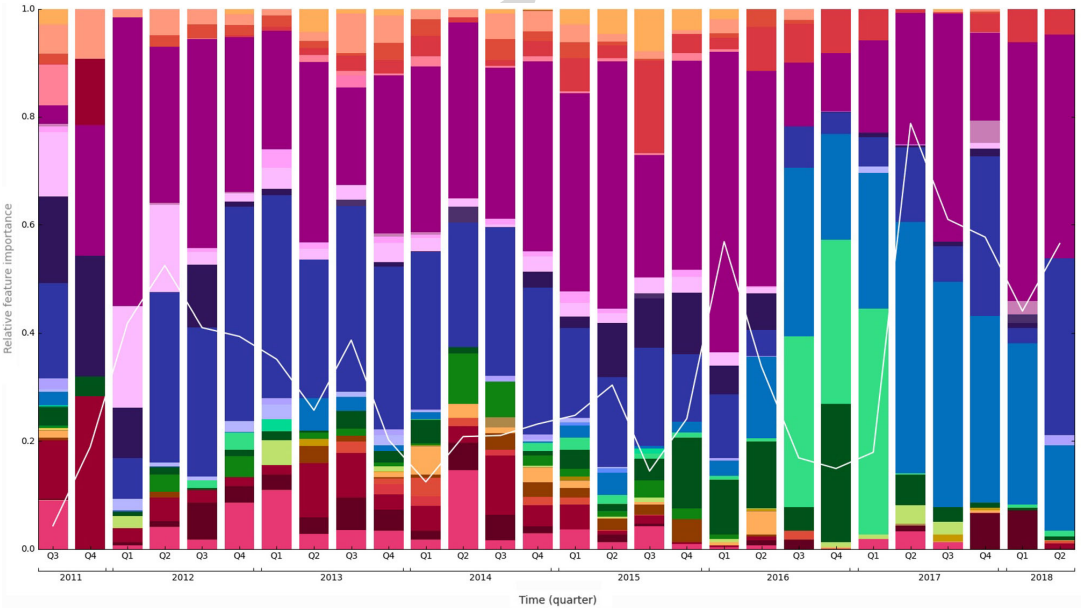


Fig. 5 Importance for specificity, calculated for the reduced feature set

SPI Journal: 11416 MS: 0432 TYPESET DISK LE CP Disp.:2022/8/22 Pages: 32 Layout: Large

Figure 5 clearly illustrates the existence of concept drift in benign data. As can be observed, READ_PHONE_STATE permission (i.e., dark blue), which was very relevant in the period from 2012-Q2 to 2016-Q1, lost its significant role in the period encompassing from 2016-Q2 to 2017-Q3, being outperformed by MOUNT_UNMOUNT_FILESYSTEMS (i.e., light blue) in that specific time frame. More precisely, MOUNT_UNMOUNT_FILESYSTEMS emerged as the most important feature for that period and kept its significant role after it. Furthermore, for a few quarters, INTERNET permission (i.e., light green) surged as a locally important feature. Beyond 2017-Q3, READ_PHONE_STATE importance role surged back again after the weakening period. These *sudden* changes in the importance of features may have contributed to the sharp decrease in performance observed in 2016-Q3, as shown in Fig. 3). Besides, the sudden change is correlated with a decrease in the maximum absolute importance (i.e., a major dip in the white line), coincidental with the local dominance of the INTERNET permission.

For the malware recognition task, depicted in Figs. 6 and 7 for the extended and reduced feature sets, respectively, the situation is significantly different. In this case, the importance of features changed dramatically over time, suggesting a more complex and abrupt *drift* in the data. More specifically, READ_PHONE_STATE and SEND_SMS were the most important features in the 2012-2013 time frame. After it, SEND_SMS rarely became significant, whereas READ_PHONE_STATE kept its significant role for almost all the analyzed time frame. In the period from 2014-Q1 to 2014-Q3, a sudden change in the important permissions is observed, especially in 2014-Q2 when permissions not included in the reduced set emerged as critically important (i.e., 50% of total importance). A fact that was repeated in 2017-Q1. From 2014-Q4, READ_PHONE_STATE surged back as the most important feature, showing decay over time and finally being of minimal importance from 2017. Thus, the opposite situation to the specificity case is observed. Lastly, from 2017-Q2, a similar pattern is spotted on both recall graphs, with a larger number of features sharing the status of important features and similar relative importance values.

A remarkable fact in the recall case is that, as opposed to specificity, the absolute magnitude of the most important feature per quarter is small, never surpassing 0.25 or 25%, and with an average value of around 4.8% in both feature sets.

The observed concept drift for recall when the reduced feature set is used is similar to the extended feature set case but more evident in the last periods, as shown in Fig. 7. More accurately, until 2017-Q1, READ_PHONE_STATE permission was consistently the most critical feature. After that period, this feature was marginalized, emerging as relatively important in just two periods, in quarters dominated by SYSTEM_ALERT_WINDOW and RECEIVE_BOOT_COMPLETED permissions, among others. Despite that, the overall trends and most important permissions are consistent between both feature sets but show different overall importance orderings (i.e., READ_PHONE_STATE, SEND_SMS, WAKE_LOCK, ACCESS_WIFI_STATE, RECEIVE_BOOT_COMPLETED and ACCESS_NETWORK_STATE). However, the influence of the extended features for recall appears to be much more significant in specific periods than in the specificity case. For instance, in 2014-Q2 and 2017-Q1, the importance is almost evenly split between the reduced set features and the extended set features. A fact not observed in the specificity case, with smaller grey areas in Fig. 4 compared to Fig. 6.

Overall, when both detection tasks are compared, more features appear to be important for the malware recognition task, with the most important feature per quarter reporting lower absolute importance for the recall task. Therefore, the complexity of the recall task is deemed as more challenging and varied, less stable over time, and more susceptible to sudden concept drift. The observed changes in the last periods demonstrate that malware is more unpredictable than benign software in permissions usage.

To further explore these differences in important features, the results obtained for specificity and recall were compared for each quarter using *Wilcoxon signed-rank* test [77]. The statistical comparison, detailed in Appendix B, confirmed that the important features for the specificity task are not equally relevant for the recall task.

For the sake of completeness of the characterization analysis, Figs. 8 and 9 unveil the important features behind the grey areas in Figs. 4 and 6, respectively. More specifically, Fig. 8 provides the features belonging to the extended feature set that were found relevant in the analyzed time frame for the specificity task, while Fig. 9 provides the same information for the recall task. In these graphs, the features belonging to the reduced feature set are hidden behind the grey areas. Even though the importance of these features has been demonstrated to be significantly lower than the ones included in the reduced feature set, their analysis enables us to fully characterize the evolution of the importance of permissions for the whole analyzed time frame, from 2011 until 2018.

In the specificity case, depicted in Fig. 8, only 21 features added in later releases of Android OS were found relevant at some specific period in the analyzed time frame. As can be noted in Appendix C, the available permission set was refined in almost every Android release, extending the *available* permissions set to 157 permissions at the time of writing. More specifically, WRITE_EXTERNAL_STORAGE, BIND_DEVICE_ADMIN, and READ_EXTERNAL_STORAGE were identified as the most relevant features for specificity from the later additions to the feature set. Besides, Fig. 8 evidences locally emerging concept drift as some spe-

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

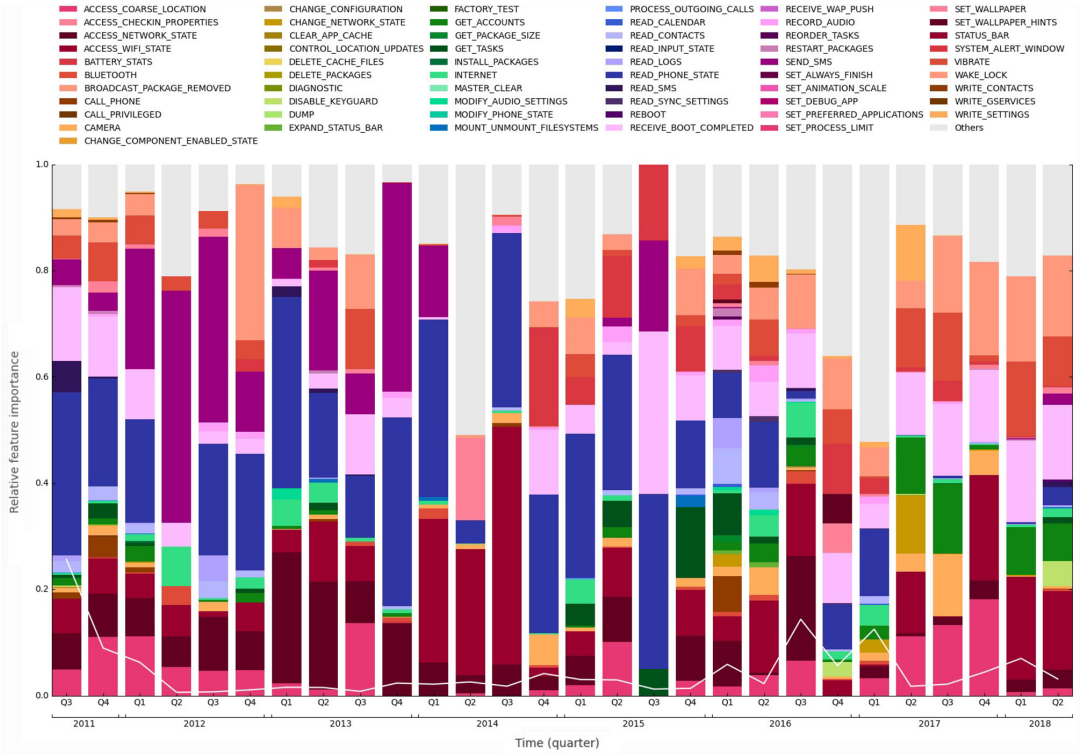


Fig. 6 Importance for recall, calculated for the extended feature set

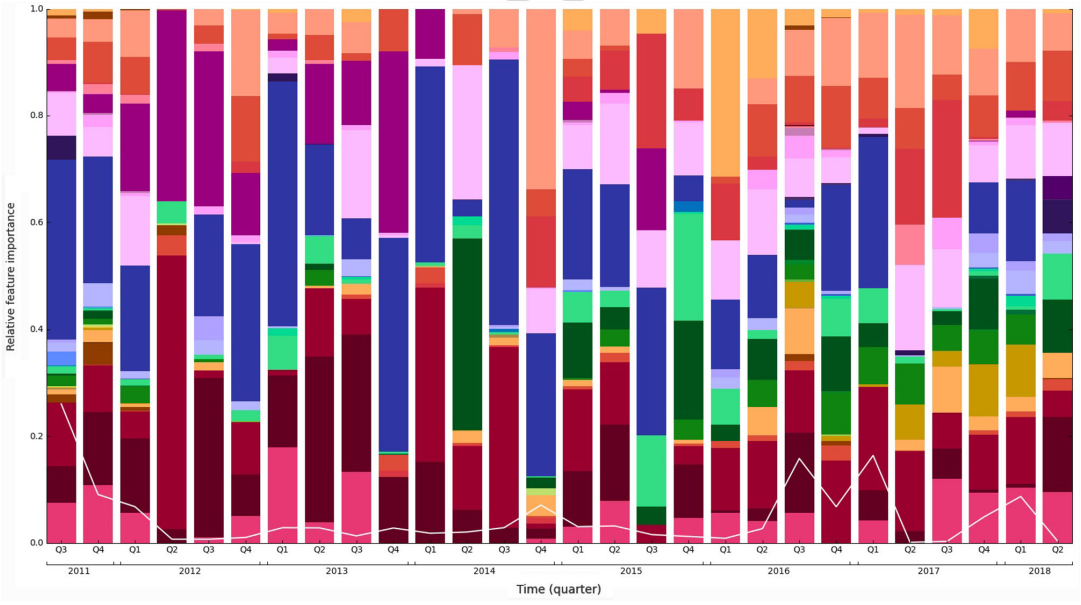


Fig. 7 Importance for recall, calculated for the reduced feature set

SPI Journal: 11416 MS: 0432 TYPESET DISK LE CP Disp.:2022/8/22 Pages: 32 Layout: Large

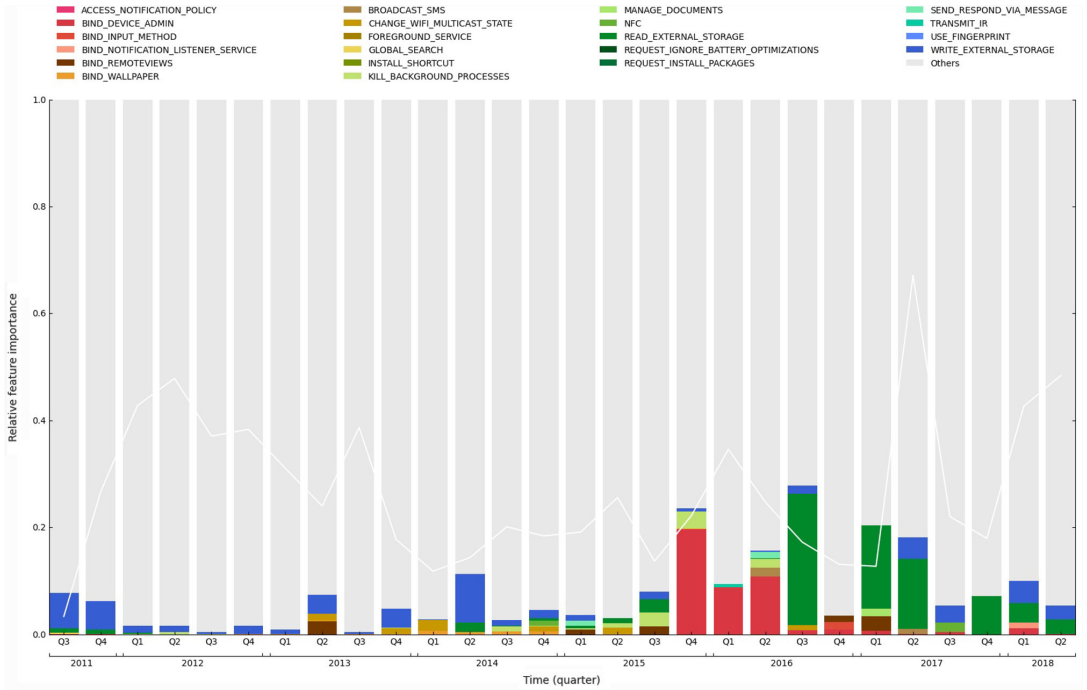


Fig. 8 Importance for specificity, showing only the extended features

cific features are relevant in some quarters but not globally (e.g., BIND_DEVICE_ADMIN from 2015-Q4 to 2016-Q2).

For the recall task, 27 features belonging to the extended feature set were found important at some point in the analyzed period, as provided in Fig. 9. More interestingly, in some cases, such as 2014-Q2 and 2017-Q1, an *extended* feature became the most important for the specific quarter. This fact was not observed for specificity and demonstrates the greater complexity of the malware recognition task, which is characterized by more sudden and, consequently, complex to handle concept drifts. In addition to the relevant features found for the specificity case, KILL_BACKGROUND_PROCESSES and BIND_WALLPAPER were of remarkable importance for the malware recognition task in specific quarters. Besides, WRITE_EXTERNAL_STORAGE showed an even more significant impact on recall, present in almost all quarters with distinctive relative importance.

Lastly, when Figs. 8 and 9 are compared, the observed concept drifts provide an interesting observation. For some prominent permissions, the features were found relevant for a task during a specific time frame, and then, after losing their importance for that task, they became relevant for the other task. For instance, BIND_DEVICE_ADMIN was important firstly for specificity (i.e., 2015-Q4 to 2016-Q2), and immediately after its importance vanished for this task, it emerged as an important feature for recall (i.e., 2016-Q3 to 2017-Q1). The opposite is observed for KILL_BACKGROUND_PROCESSES, becoming first relevant for recall and later for specificity.

In summary, the characterization analysis performed evidenced the critical importance of the initial set of permissions to build an effective recognition system, the lower relevancy for such a purpose of the later added permissions, and that even though concept drift issues were found in benign and malware data, the former shows relative stability with gradual changes being relatively easy to address, whereas the latter is characterized by more sudden, complex concept drifts dominated by specific features, making it harder to handle. Besides, the set and degree of importance of features differ for both tasks. Therefore, the analysis performed in this section evidences the dynamism and constantly evolving nature of the malware threat landscape and emphasizes the critical requirement to address concept drift for any solution aiming to provide long-lasting effective malware detection and adapt, updating its knowledge in an ever-evolving threat landscape. This constantly changing nature has been overlooked by all the proposed detection solutions using permissions in the related literature.

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

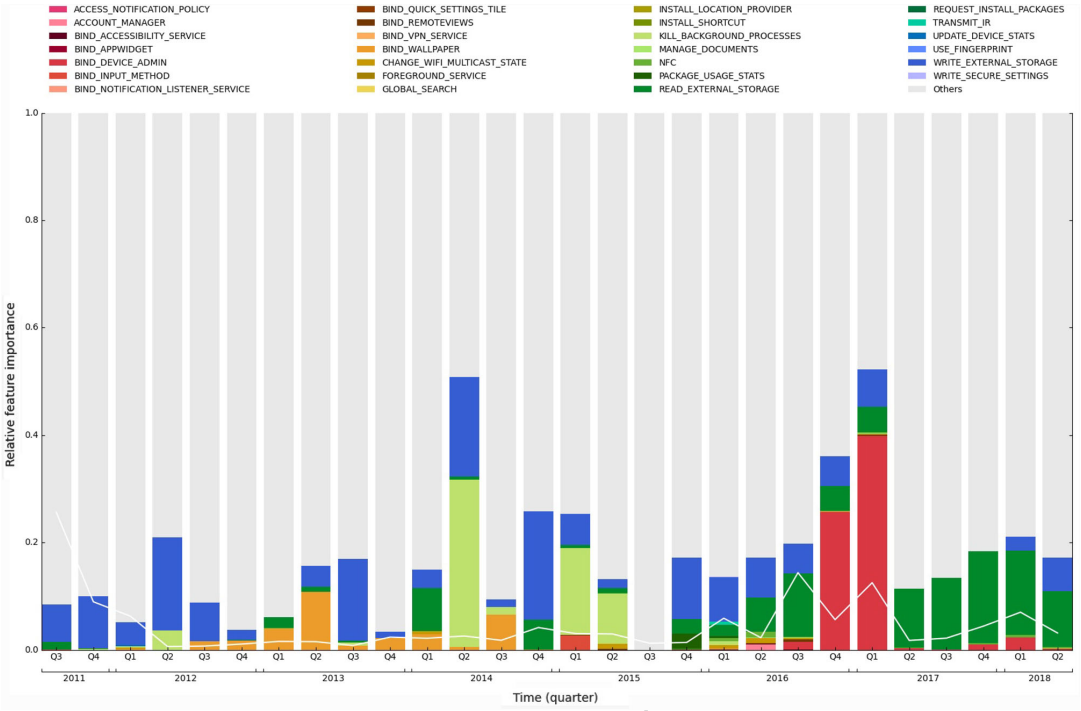


Fig. 9 Importance for recall, showing only the extended features

5.3.2 Class-based recognition performance

The F1 score metric, reported in Fig. 3, is a relevant performance metric to assess the effectiveness of malware detection solutions. However, class-based recognition comes in handy to assess the effectiveness of the detection solution to detect either of the classes and evaluate the degree of precision in the forecast of benign data.

The effectiveness of the model to detect each of the classes is provided in Fig. 10. This graph compares recall, f1 score, and specificity metrics for the reduced and extended feature sets.

More specifically, the red lines provide quarterly recall information for the full (i.e., red solid line) and reduced feature sets (i.e., red dashed line). The horizontal red lines provide the average recall performance for the whole time frame for both feature sets. Similarly, the green and light grey lines provide the same information for specificity and F1 score, respectively.

As can be noticed in Fig. 10, the model provided consistently high specificity quality (i.e., over 0.8) using either of the feature sets. This fact emphasizes the effectiveness of the permission-based model to recognize benign apps effectively over time and is consistent with the *smooth* concept drift that characterized these data.

However, the situation is notably different for the recall metric. Even though the average recall performance of the system shows an accuracy over 0.90, even reaching 0.99 in 8 periods, the malware recognition performance dips in two specific time frames, as evidenced by the F1 score and recall performance metrics in Fig. 10. The system failed to identify new malware samples effectively in those specific chunks but kept high benign software recognition performance. Figure 10 shows that the reduced feature set is more suitable for long-term accurate malware detection than the full feature set, as the average line for the full feature set is below the average line for the reduced feature set. More precisely, the average recall value for the reduced feature set is 91.7%, whereas for the extended feature set is 89.9%. Besides, the third dip was less severe when the reduced feature set was used. Therefore, for malware detection purposes, the reduced feature set is preferred.

The opposite situation happens in the case of benign software recognition, where the extended feature set provides better average performance than the reduced feature set. More specifically, an average of 91% specificity is obtained with the extended set, whereas for the reduced feature set,

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

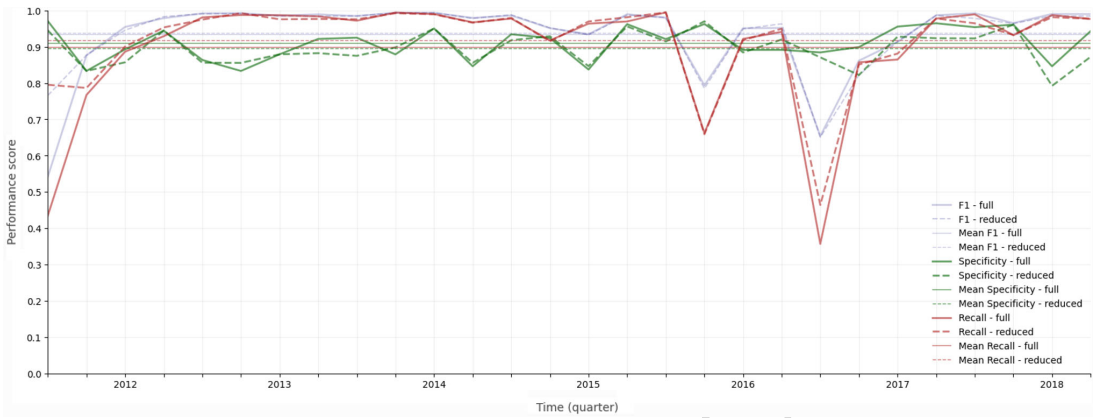


Fig. 10 Comparison of model's performance metrics using both feature sets

the average value stands at 89.5%. This observation is consistent with the findings in [53], where benign data samples were found to use a smaller but more varied set of permissions than malware apps. Consequently, the extended feature set, which includes more permissions, provides better overall performance for this task.

Despite the overall high performance of the detection system on both tasks, the system provided diminished malware detection performance, with different degrees of severity, at three specific time frames, namely, 2011-Q3 (initial period), 2015-Q4, and 2016-Q3. The cause and related insights behind these exceptional cases are described in the following paragraphs.

The first dip happens in the initial period, 2011-Q3. Even though this should not be considered a dip in performance, it is worth analyzing its cause. In the initialization phase of the system, where the model has access to a limited amount of data, the performance of the system strictly depends on the generalization capabilities of the initial data chunk regarding the following chunks/quarters. Therefore, the initial performance directly relates to the quality of the data in the initial chunk and not to the learning capabilities of the system, as there is no previous reference of detection performance. In our case, the initial chunk was split into n ordered data chunks, where n refers to the number of classifiers in the pool. The system was initialized in this initial quarter, building a distinct model with each data chunk and incorporating it into the pool. As a result, the low initial performance is likely caused by an insufficient variety of initial data to capture the phenomenon accurately. In this regard, the initial chunk was significantly dominated by benign apps (i.e., 97.25%), thus limiting the learning capabilities for an effective malware detection performance. The specificity performance tops in this initial quarter. However, even in this challenging learning situation, where the proportion of malware was very small (i.e., 3.75%), the system provided acceptable performance when the reduced feature set was used (i.e., 0.77 F1 score). The malware detection performance increased in the subsequent quarters, demonstrating the capabilities of the model to learn and adapt over time, even when a distinct combination of features emerged as important in close quarters (e.g., 2012-Q2). In the early quarters, the reduced set of permissions showed prominent importance, as evidenced by the high dependency of the extended model on the reduced set of permissions in the first 11 quarters, with importance of over 80% in all of them, as shown in Fig. 6.

The second and third performance drops correspond with actual concept drift.

The second performance drop, which is less severe than the third, is located in the fourth quarter of 2015. At first glance, the feature importance pattern looks similar to the previous data chunks. However, the distribution of 2015-Q4 changes significantly from the previous four chunks, with more features sharing the *important* status and a significant decrease in the domination of READ_PHONE_STATE, as can be observed in Figs. 6 and 7. In addition, the absolute importance value reaches a local minimum in that quarter, which emphasizes the fact that more features are important for the model, and none of them dominates significantly. This also includes the extended features (i.e., Fig. 9). An additional contributing factor might have been the special characteristics of the previous chunk, which shows a radically different importance distribution, dominated by a small number of features from the reduced feature set and no *extended* feature showing importance. Thus, all these changes in feature importance distribution evidence a light drift in this quarter's data concerning previous malware data, which can be seen as promoting factors behind this moderated performance dip.

This malware data shift is coincidental with the specificity score reaching its maximum value and being affected

SPI Journal: 11416 MS: 0432 TYPESET DISK LE CP Disp.:2022/8/22 Pages: 32 Layout: Large

by a decrease in the importance of previously important features, as shown in Fig. 4, and the outbreak of BIND_DEVICE_ADMIN as an important feature. This situation, where the model had to learn about significant changes in benign software, might have also impacted the recall performance at the expense of boosting specificity. Despite all these learning challenges, the F1 score of the model showed an acceptable performance (i.e., over 0.80), improving in the subsequent chunks.

The third drop is located in the third quarter of 2016. In this quarter, recall dips dramatically, especially for the extended feature set. At first sight, this period shows remarkably different characteristics from the previous ones. Firstly, it shows a significantly different feature distribution. In this period, 35 features were found important for the extended feature set (i.e., 16 belonging to the reduced set), whereas, in the previous quarters, fewer than 20 features were found important, on average. This relates to a change in data complexity, as the model had to learn and rely on more features for proper detection. Secondly, the dominant feature in the previous chunks, READ_PHONE_STATE, was relegated to a marginal role, and two features emerged as dominant features, ACCESS_NETWORK_STATE and ACCESS_WIFI_STATE. Besides, an increase in READ_EXTERNAL_STORAGE, belonging to the extended feature set, was also observed. The emergence of new important features correlated with an increase in absolute importance value, reaching a local maximum, as depicted by the overlay line. Thirdly, BIND_DEVICE_ADMIN diminished its importance significantly for specificity and emerged as important for recall for the first time, as shown in Fig. 9. Lastly, the sudden concept drift in recall is combined with a significant drift in specificity as new features irrupt as relevant with remarkable importance values in the reduced feature set (i.e., INTERNET and MOUNT_UNMOUNT_FILESYSTEMS) and the extended feature set (i.e., READ_EXTERNAL_STORAGE), as shown in Figs. 4 and 8, respectively.

In this case, for both recall and specificity, the changes were too sudden, unexpected, and distinct from previous data patterns for the classification pool to be able to deal with the new data characteristics properly. However, after this sudden and severe decrease in the recall performance, where the system still kept acceptable recognition capabilities for benign software, the pool adapted and learned from the new data, improving and recovering past performance levels in the subsequent chunks. No significant dip in performance was observed again.

In summary, the analysis performed in this subsection enabled us to correlate the quarterly characterization performed in the previous subsection with class-based performance results and find the rationale behind the decrease in performance in some specific quarters. The analysis evidenced that concept drift handling of benign data is an easier task due to the smoother feature variability in these data provoking a gradual concept drift and that sudden drifts in feature importance observed for malware data correlate with performance dips. The more dramatic and unexpected the changes, the more severe the decrease in performance observed. This fact is augmented when the extended feature set is used. The reduced feature set shows improved performance and less severe dips for malware data. Regarding specificity, even though both feature sets work well, the more varied usage of features by benign applications makes the extended feature set occasionally outperform the reduced feature set.

5.3.3 Malware family evolution analysis

This subsection aims to explore the third dip in detail, seeking the etiology of the unprecedented complexity observed in this quarter and the significant decrease in performance that occurred.

Android malware is constantly evolving, and this is reflected in the prevalence of malware families over time. Figure 11 shows the distribution of the top 10 malware families per quarter in the time frame from 2011-Q3 to 2018-Q2. The graph shows the prevalence of 54 malware families over time, indicated using specific colors for each one. Besides, the graph reports the number of malware families per period (i.e., white stars in the middle of each bar, whose values are related to the right vertical axis). The F1 performance of the detection model is provided with a white dashed overlay line.

As can be noticed, the major dip happens in a quarter dominated by the *Slocker* malware family, the first and most relevant Android ransomware family (i.e., 68% of the samples in 2016-Q3). This fact suggests that the dip might have been directly caused by diminished ransomware detection capabilities. However, it is worth noticing that ransomware was also dominant in 2015-Q3 (i.e., 53% of the samples), and the detection model provided over 98% F1 score and higher recall, as shown in Fig. 10.

To better explore the phenomenon, one-class anomaly detection models were built for the most prevalent families. Instead of using the label as the concept of class to build the one-class anomaly models, the malware family was leveraged as the class construct. The main idea behind these *one-class* or, in our case, *one-family* anomaly models is as follows. If an effective one-class anomaly detector for a specific family can be built in a specific quarter (i.e., high performance on the training data), when tested with samples of the same malware family in subsequent chunks, it should be able to detect them properly, reporting high accuracy. As a result, if the family samples change their character (i.e., malware family evolution), the anomaly model would reflect it as a performance decrease. Otherwise, if malware does not change significantly, high accuracy is expected.

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

Fig. 11 Distribution of malware families per period

Local Outlier Factor (LOF) algorithm was used to build the *one-family* anomaly detection models. LOF allows detecting *dissimilar* data samples using local density deviation of data points with respect to their neighbors [78]. Figure 12 provides *one-family* anomaly models for 12 predominant families in the time frame ranging from 2011-Q3 to 2018-Q2. The initial models for each family were induced in different quarters when the malware family produced an outbreak, and they were tested with subsequent chunks until the malware family's presence vanished.

Figure 12 provides relevant insights about malware families' evolution, which are directly related to the performance dips and concept drift observed. Firstly, most malware families showed similar or the same character over time. This is especially evident in the first half of the analyzed time frame, where almost all malware families showed the same features over time, as evidenced by the high accuracy values provided by each initial anomaly model tested with the subsequent quarters. It implies that most malware families do not evolve much regarding permissions over time, thus making these features powerful discriminators. Secondly, and more interestingly, the *Slocker* family did not follow that pattern. The initial model built for the *Slocker* family dips significantly in 2016-Q3, showing a dramatic and sudden change in characteristics concerning the initial model samples. The experimental findings are confirmed by [79], which reported that in the second half of 2016, over thrice Slocker variants were detected in comparison with the same period in 2015, and by [80], which reported about a recursive ransomware outbreak characterized by evolution into a more sophisticated and diverse malware family. This increase in variety and sophistication was already suggested by the more complex and diverse quarter characterization depicted in Fig. 7.

Additional evidence of the diversification of the *Slocker* ransomware family is provided in Fig. 13. Figure 13a provides individual predictions' *explanations* for samples belonging to the *Slocker* family in 2015-Q3, whereas Fig. 13b provides the same information for *Slocker* samples belonging to 2016-Q3. The comparison between the 2015-Q3 and 2016-Q3 samples is significant due to their similarity with regards to the large dominance of this family in these quarters and the fact that they yielded notably distinct detection performance. *Shapley values* are leveraged in these graphs to explain the reasoning behind individual predictions by the

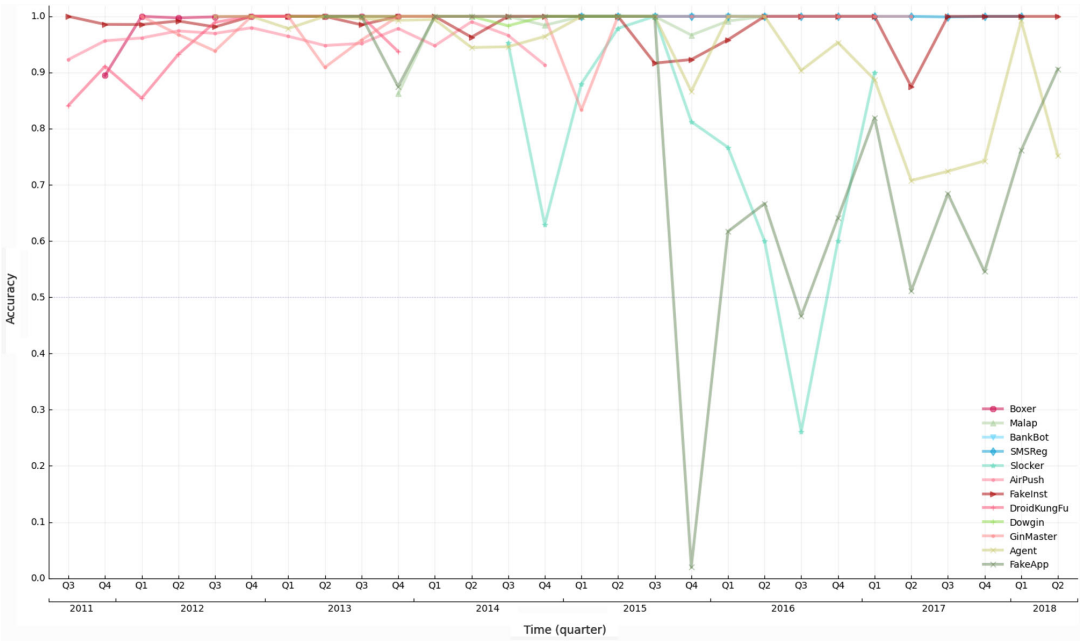
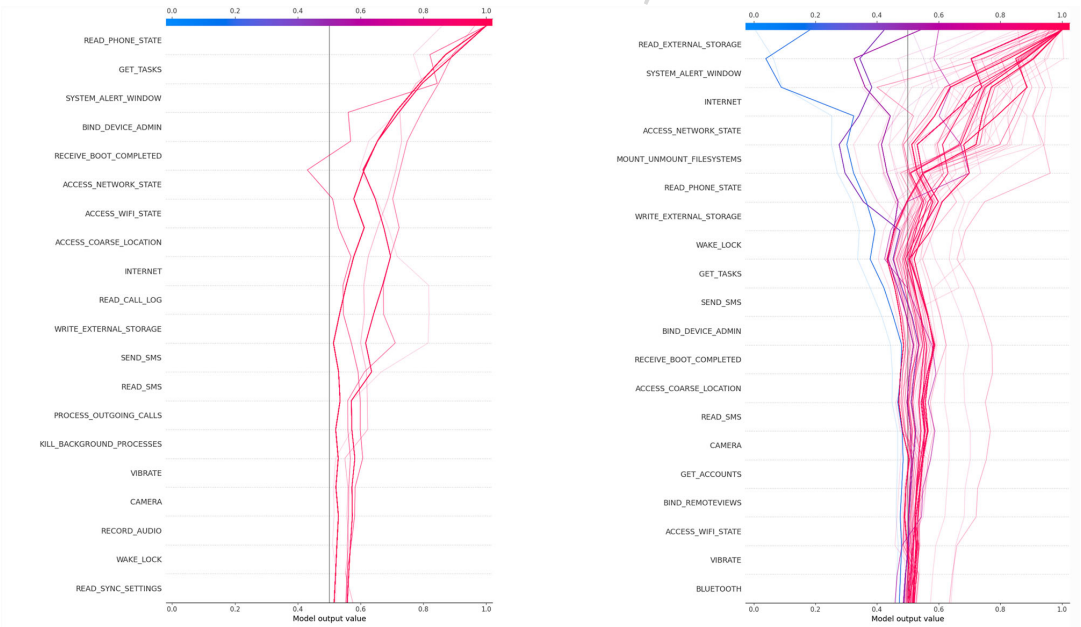


Fig. 12 Performance of *one-family* anomaly detection models



(a) 2015-Q3 Slocker predictions decision paths

(b) 2016-Q3 Slocker predictions decision paths

Fig. 13 *Slocker* family 2015-Q3 and 2016-Q3 predictions' decision paths

SPI Journal: **11416** MS: **0432** ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:**2022/8/22** Pages: **32** Layout: **Large**

model. The graphs should be interpreted as follows. The initial probability of a sample belonging to the malware class is 0.5. As the data set is balanced, it reflects the expected value. On the left vertical axis, and from bottom to top, the features are ordered in increasing importance for the final decision made by the classifier, specified as prediction probability on the top part of the graph. The range, from 0 to 1, refers to the malware class probability. The lines or *decision paths* starting from the bottom, and reaching a prediction probability in the top part, give a notion of the impact of the features to classify each sample regarding the final probability. As all the data samples used to construct these graphs were Slocker samples, the expected classification result was over 0.5 in all cases (i.e., the model identified it as malware). The color of the line is based on the final prediction, highlighting the paths leading to the *malware* prediction (i.e., red) or the *no malware* prediction (i.e., blue). The more intense the line color, the larger the number of samples that followed that decision path. Both graphs use the extended feature set models as they enable us to fully understand the differences between the important features in each case.

As can be seen in Fig. 13a, all ransomware samples were correctly classified by the model, and the explanations are similar for all samples, with just six decision paths explaining all of them. Therefore, the ransomware samples in this quarter show similar characteristics, and consequently, similar features are used to classify them. On the contrary, Fig. 13b shows numerous paths used to classify the ransomware samples, with varying features importance, and many misclassified samples reach the no malware outcome. Therefore, Slocker samples from 2015-Q3 are significantly distinct from 2016-Q3 samples. The samples from 2016-Q3 are more varied, showing many different characters, and pose a more complex threat than the initial Slocker samples. In this regard, the differences in important feature sets on the explanations provide useful insights into the changes in behavior and main characteristics of these samples. For instance, the most important permission to discriminate 2016-Q3 ransomware was READ_EXTERNAL_STORAGE, which provides read access to files outside the app-specific directory and the media store (i.e., probably to access sensitive information and send it to the C&C server), whereas, in 2015-Q3, the ransomware samples were more concerned about the running tasks and phone general information. These plots evidence the significant change in the Slocker family, disguised in many forms to avoid detection [80], and, consequently, the added complexity for effective discrimination in the 2016-Q3 quarter.

Lastly, Fig. 12 also provides relevant insights about the reasons behind the second performance dip, located in 2015-Q4. As can be observed, the quarterly malware distribution graph, depicted in Fig. 11, shows a significant increase and domination of the *FakeApp* family (i.e., 29.73%) in this period. At the same time, the related *one-family* anomaly detection model for that family shows that almost all samples for that quarter were detected as *anomalies*, thus indicating an *extreme* change in this malware family.

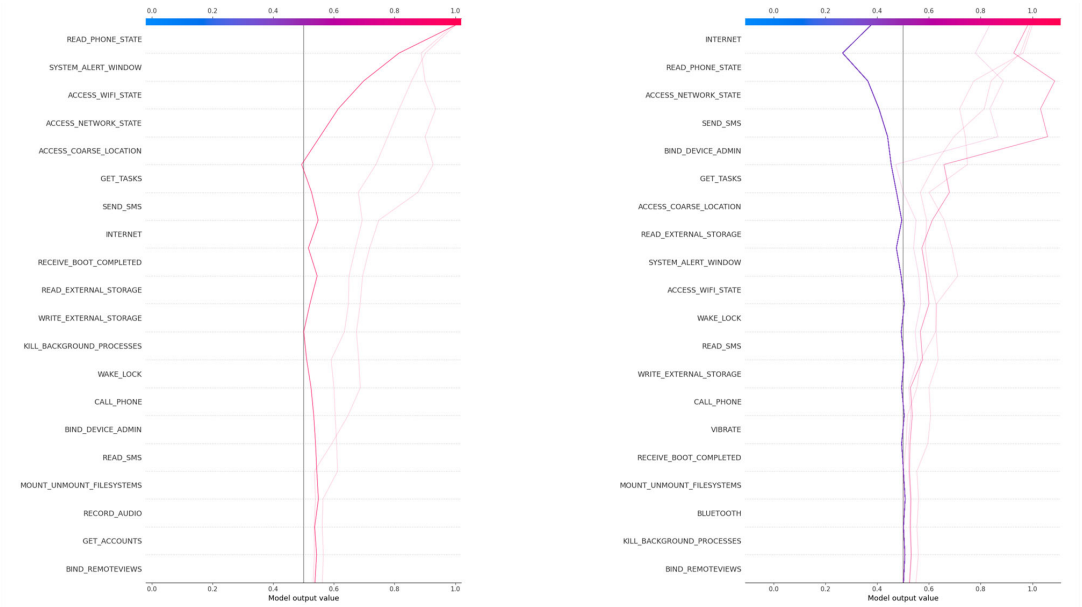
Similar to the ransomware case, Fig. 14 shows the decision paths for the *FakeApp* samples belonging to 2015-Q3 and 2015-Q4. More precisely, Fig. 14a provides explanations for *FakeApp* samples belonging to 2015-Q3, whereas Fig. 14b provides the same information for the *FakeApp* samples from 2016-Q4. As can be observed, a similar but more extreme outcome than in the ransomware case is depicted in Fig. 14, where the majority of data samples belonging to 2015-Q4 were misclassified. However, they showed consistent characteristics, and therefore only a few decision paths are observed.

The malware family analysis performed in this subsection enabled us to compare the performance dips and the changes in the malware threat landscape. It has been shown that malware family evolution is behind the performance dips and that sudden outbreaks and evolution of specific malware families explain the results observed in previous sections.

6 Discussion

The present study focused on the analysis of permissions evolution over time and its significant impact on malware classifiers built using these data. The proposed method, built exclusively using permissions and an ensemble of classifiers selected from a pool, allowed it to *learn* and adapt to concept drift, providing high performance over an extended period. Despite the decrease in performance observed in 2015-Q4 and 2016-Q3, an average F1 score of 0.93 was achieved in the 2011-Q3 to 2018-Q2 time frame. This consistent high-performance emphasizes the effectiveness of the proposed solution to deal with Android malware concept drift using permissions as sample descriptors. No prior solution using just permissions has addressed the concept drift issue. Therefore, this study, which directly tackles this issue, constitutes a novelty in the field with no comparative reference. The studies in the literature that used permissions before, achieving high-performance metrics, were built and tested using small and *outdated* snapshots of data from Android history. Only [17] included long-term data (i.e., 2010-2016) achieving over 0.90 F1 score. However, the usage of random selection to generate the training/testing sets, thus neglecting concept drift, poses severe concerns about the generalization capabilities of this detection system. Our study shows that if the concept drift issue is addressed, permissions alone can provide long-term effective malware discriminatory capabilities. This allows leveraging their security-related implications to enhance and *understand* the detection systems. Notwithstanding that, the usage of permissions with

SPI Journal: 11416 MS: 0432 TYPESET DISK LE CP Disp.:2022/8/22 Pages: 32 Layout: Large



(a) 2015-Q3 FakeApp predictions decision paths

(b) 2015-Q4 FakeApp predictions decision paths

Fig. 14 *FakeApp* family 2015-Q3 and 2015-Q4 predictions' decision paths

additional dynamic and static features may enhance the detection capabilities of the proposed system, especially in those time frames where permissions alone may struggle to detect effectively under sudden concept drift (e.g., 2016-Q3). This constitutes part of our future work.

The experimentation performed in this research has shown that the initial set of permissions, released with the first version of Android OS, is currently the most relevant feature set to discriminate malware apps. The later additions to the permissions set, while showing relatively high local importance in specific periods, do not show the powerful and long-term consistent discriminatory capabilities provided by the initial permission feature set. This fact has remarkable implications in the generation of permissions-based effective discriminatory models as there is no *actual* need to update the feature set after every release, so the maintenance requirements of the detection system diminish significantly. Even though this may change in the future, as the initial set of permissions covers core security aspects (e.g., access to sensitive data), if they are not redefined or substituted in future API releases, they might retain their discriminatory power and be an essential part of the arsenal for effective malware detection. Besides, the *curse of dimensionality*, a degenerative performance issue related to ever-growing feature sets in machine learning models, is avoided when a small feature set is used. Furthermore, permissions are the easiest static features to collect and are inherently *interpretable* as they have security-related implications behind them. This fact makes the machine learning-based detection models based on permissions computationally efficient, and their outcomes easily explainable and traced using *post-hoc* explainers or interpretable machine-learning models.

The analysis of the importance of features and their relation to malware distribution has provided relevant insights to explain the dips in performance of the proposed permissions-based detection system. Permutation feature importance and Shapley values have been used to provide permissions characterization over time and a better understanding of the decision outcomes, respectively, thus providing useful insights about malware evolution for its effective detection. None of the previous permission-based proposed solutions took into account concept drift in their models nor provided any thorough characterization of permissions evolution over time.

7 Limitations and threats to validity

The limitations that may challenge the generalization of the results are centered around the data set, model performance, and interpretation methods used.

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

The quality of the data is critical to building effective machine learning-based detection systems. Thus, for an accurate malware detection solution to perform effectively, the data used should be representative of the actual situation. This is especially critical for production setups. The data set used, *KronoDroid*, provides a good approximation to the malware threat landscape in the period studied but may have limitations regarding the temporal accuracy, which is critical for proper concept drift study, and the quantity of data. These facts may threaten the external validity of the experimentation. However, no other data set for Android research includes timestamps nor suits for the evaluation of long-term evolution. Furthermore, the naming for malware families is not homogeneous among vendors, which may impact the labeling and, consequently, the range of distinct behaviors included under the same family label. Despite that, the findings using *KronoDroid* data have shown a correlation with the actual Android historical timeline events, thus showing the goodness and reliability of the data set.

Regarding the model performance, the time frame used to split the data (i.e., quarters) was found experimentally, based on the characteristics of the data set. The solution provided high performance for an extended period. However, two dips in performance were observed due to sudden concept drift events. In a real-world setup, where data is constantly flowing, the time constraints should be narrowed down (i.e., experimentally find the optimal chunk size/length), and the sudden concept drift would likely be handled more efficiently as *smaller* data chunks should minimize and smooth out the concept drift impact. Despite these limitations, the solution provided high specificity, recall, and F1 metrics for a seven-year-long time frame. They can be used as a reference of the capabilities of the system for production setups.

Finally, the interpretation methods used to characterize the concept drift and explain the individual decisions, which were permutation feature importance and Shapley values, have inherent limitations, especially when correlated features are included [75]. To minimize this issue, the correlation between features was analyzed in the data pre-processing step, and correlated features were removed. Despite that, no machine learning interpretation method is free from limitations, and the quality of the data used can also significantly impact the outcome [75]. Therefore, any interpretation method output should be carefully considered. In our case, the global interpretability method (i.e., permutation feature importance) and the local interpretability method (i.e., Shapley values) provided similar insights about the data, supporting our findings.

8 Conclusions

Permissions are *first-line* security constructs within the Android ecosystem. The analysis of their evolution and usage can provide relevant highlights about the malware scene, especially when analyzed long-term.

We leveraged all these features in this study. Our results show that when concept drift is addressed, permissions alone can be used to build a long-lasting, effective malware detection solution. An average F1 performance of over 0.93 was achieved in data that span seven years. Furthermore, permissions, which have been relegated to a secondary role in Android malware detection systems, can provide powerful insights about the behavior of apps, as they are inherently *comprehensible* features, a property that can be used to enhance detection systems and malware knowledge.

This research shows that a reduced number of features can be used to build an effective system, especially for the specific malware recognition task, and that there is no need to update the feature set after every Android release. The analysis of the evolution of permissions shows that malware is a complex phenomenon in constant evolution, so concept drift issues must be considered to keep high detection performance. However, clear patterns in permissions data are observed, especially for the benign recognition task, which can be leveraged to build more effective systems.

Family-based analysis of permissions evolution and its characterization is part of our future work.

Appendix A. Two-sample Kolmogorov-Smirnov statistical test

To formally compare the performance of both feature sets (i.e., extended vs. reduced), the non-parametric *Kolmogorov–Smirnov* test was used to analyze the equality of both probability distributions (i.e., two-sample K–S test). The test statistic quantifies the distance between the empirical distribution functions (EDF) of two data samples. In our case, the distribution of F1 score data is compared for the reduced and extended feature sets. The results show that with a p-value = 0.7634, the test cannot be the basis for H_0 rejection that the F1 score distributions come from the same distribution. Moreover, the K–S statistic value, which reflects the maximum distance between the EDFs, is relatively small (i.e., K–S = 0.04). The test cumulative density function (ECDF) is illustrated in Fig. 15. Thus, it can be concluded that there is no significant difference in detection performance between the compared feature sets.

SPI Journal: 11416 MS: 0432 TYPESET DISK LE CP Disp.:2022/8/22 Pages: 32 Layout: Large

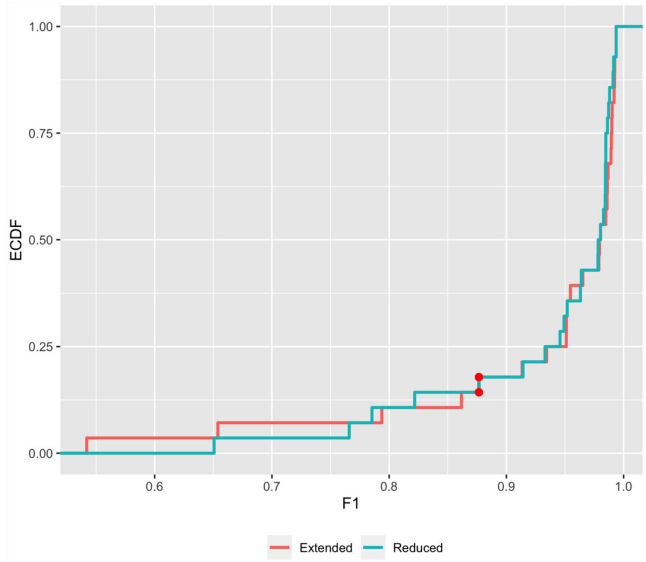


Fig. 15 Kolmogorov-Smirnov test to compare F1 for the reduced and extended feature sets. The maximum distance between EDFs is 0.04 (marked in the graph with red dots).

Appendix B. Wilcoxon signed-rank test

Wilcoxon signed-rank test is a non-parametric statistical hypothesis test (i.e., no normality distribution is assumed) that enables the comparison of two data distributions (i.e., specificity and recall in this case). However, the test requires the same amount of data on both distributions and, in our particular case, as not all features were found important in all periods, there were missing values in quarters regarding specific features. The quantity of missing values is critical. The data for the reduced vector showed nearly 70% missing values while reaching 83% for the extended vector. Therefore, the statistical analysis was performed only on the reduced feature set data. In this regard, as the negative values of function (4) are unknown but are needed for the test, a solution could be replacing missing values with zero values. Yet, the large missing value ratio might bias the comparison results, groundlessly increasing the similarity of the vector with a large number of missing values. Therefore, a better approach is to replace the missing values with the mean importance of the feature, calculated for the given data set and taken as a negative value. Thus, vectors are compared using means when the number of missing values is high and using the distribution of importance otherwise.

The results of the Wilcoxon test are reported in Table 2, calculated for the reduced feature set, and used to distinguish permissions important to optimize specificity and recall tasks. The table reports the features with a p-value less than 0.005, which suggests a highly significant difference between the compared vectors. The occurrence refers to the number of not missing values in the compared vectors (i.e., the number of times the feature was found important in recall or specificity). The maximum and total importance are provided for each feature. The maximum importance reflects the maximum value of importance the feature had in any quarter, while the total importance is the cumulative importance of the specific feature in all quarters. The features are ordered based on the completeness of the data vectors (i.e., from a larger number of occurrences to a smaller number).

As reported in Table 2, among the 44 shared features found important for both recognition tasks, 29 showed statistically significant differences with a p-value < 0.005. Among these important features showing significantly distinct importance distributions for malware detection and benign software recognition, there are relevant concept drift-related features such as READ_PHONE_STATE, SEND_SMS, and MOUNT_UNMOUNT_FILESYSTEMS. Apart from the large total importance, all these features have high occurrence and the largest maximum importance values. In all cases, the obtained importance is significantly higher for specificity. Therefore, based on this table, it can be concluded that important features for benign app recognition (i.e., specificity) are not equally relevant for the malware recognition task (i.e., recall).

Appendix C. Android permission set evolution

Table 3 provides the modifications that changed the available set of Android security permissions over time. This table gives a notion of the dynamic character of the permission set

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

Table 2 Significantly different ($p < 0.005$) features used to maximize recall and specificity for the reduced feature set

	Permission	Importance		Occurrences
		Total	Max	
1	READ_PHONE_STATE	5.0541	0.5246	51
2	SEND_SMS	7.7330	0.6100	44
3	GET_TASKS	0.8402	0.1256	43
4	SYSTEM_ALERT_WINDOW	0.6956	0.1083	42
5	READ_SMS	0.8039	0.1488	28
6	MOUNT_UNMOUNT_FILESYSTEMS	3.1416	0.7879	25
7	READ_SYNC_SETTINGS	0.0510	0.0191	19
8	MODIFY_AUDIO_SETTINGS	0.0310	0.0214	16
9	BATTERY_STATS	0.0280	0.0126	14
10	READ_CALENDAR	0.0075	0.0032	13
11	CHANGE_CONFIGURATION	0.0215	0.0141	12
12	INSTALL_PACKAGES	0.0090	0.0052	12
13	PROCESS_OUTGOING_CALLS	0.0208	0.0193	10
14	GET_PACKAGE_SIZE	0.0114	0.0091	9
15	CALL_PRIVILEGED	0.0008	0.0008	8
16	CLEAR_APP_CACHE	0.0047	0.0046	6
17	WRITE_CONTACTS	0.0192	0.0085	6
18	ACCESS_CHECKIN_PROPERTIES	0.0003	0.0003	4
19	MODIFY_PHONE_STATE	0.0009	0.0006	4
20	REBOOT	0.0010	0.0010	4
21	RECEIVE_WAP_PUSH	0.0050	0.0047	4
22	SET_WALLPAPER_HINTS	0.0022	0.0021	4
23	DELETE_CACHE_FILES	0.0010	0.0009	3
24	DELETE_PACKAGES	0.0001	0.0001	3
25	EXPAND_STATUS_BAR	0.0053	0.0048	3
26	SET_PREFERRED_APPLICATIONS	0.0250	0.0250	3
27	READ_INPUT_STATE	0.0000	0.0000	1
28	SET_DEBUG_APP	0.0000	0.0000	1
29	STATUS_BAR	0.0823	0.0823	1

and its evolution throughout the whole Android lifetime. The permissions set was first defined for API level 1 (i.e., the first release of Android) and has been constantly modified since then. Table 3 provides the evolution of the *available permission set* from API level 1 to API level 30. The table is ordered chronologically based on API level release (i.e., from the oldest to the latest), and it provides API level-related information such as release date (i.e., *Date*) and *OS version* name. For each API level (i.e., rows), the added and deprecated permissions in the corresponding API level are provided in the *Added Permissions* and *Deprecated Permissions* columns. Furthermore, for each added permission, the protection level is provided in the column *Type*. Three protection levels are defined: *dangerous*, *normal* and *others*, reported as D, N and O, respectively. The *others* category refers to permissions that can be requested by third-party apps and that do not belong to the dangerous or normal category, as defined in the official documentation [21]. If the permission cannot be used by third-party apps, it is referenced with a hyphen (-). For each deprecated permission, the API level that introduced the permission is provided within parenthesis. Lastly, the column *Set* reports the number of available permissions (i.e., excluding deprecated) in each API level.

SPI Journal: 11416 MS: 0432 ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:2022/8/22 Pages: 32 Layout: Large

Table 3 Android permission feature set evolution

API	Date	OS version	Added permission	Type	Deprecated permission	Set
1	09-2008	1.0	ACCESS_COARSE_LOCATION	D		74
			ACCESS_FINE_LOCATION	D		
			ACCESS_CHECKIN_PROPERTIES	–		
			ACCESS_LOCATION_EXTRA_COMMANDS	N		
			ACCESS_NETWORK_STATE	N		
			ACCESS_WIFI_STATE	N		
			BATTERY_STATS	O		
			BLUETOOTH	N		
			BLUETOOTH_ADMIN	N		
			BROADCAST_PACKAGE_REMOVED	–		
			BROADCAST_STICKY	N		
			CALL_PHONE	D		
			CALL_PRIVILEGED	–		
			CAMERA	D		
			CHANGE_COMPONENT_ENABLED_STATE	–		
			CHANGE_CONFIGURATION	O		
			CHANGE_NETWORK_STATE	N		
			CHANGE_WIFI_STATE	N		
			CLEAR_APP_ACHE	O		
			CONTROL_LOCATION_UPDATES	–		
			DELETE_CACHE_FILES	O		
			DELETE_PACKAGES	–		
			DIAGNOSTIC	–		
			DISABLE_KEYGUARD	N		
			DUMP	–		
			EXPAND_STATUS_BAR	N		
			FACTORY_TEST	–		
			GET_ACCOUNTS	D		
			GET_PACKAGE_SIZE	N		
			GET_TASKS	N		
			INSTALL_PACKAGES	–		
			INTERNET	N		

Table 3 continued

API	Date	OS version	Added permission	Type	Deprecated permission	Set
			MASTER_CLEAR	–		
			MODIFY_AUDIO_SETTINGS	N		
			MODIFY_PHONE_STATE	–		
			MOUNT_UNMOUNT_FILESYSTEMS	–		
			PERSISTENT_ACTIVITY	D		
			PROCESS_OUTGOING_CALLS	D		
			READ_CALENDAR	D		
			READ_CONTACTS	D		
			READ_INPUT_STATE	–		
			READ_LOGS	–		
			READ_PHONE_STATE	D		
			READ_SMS	D		
			READ_SYNC_SETTINGS	N		
			READ_SYNC_STATS	N		
			REBOOT	–		
			RECEIVE_BOOT_COMPLETED	N		
			RECEIVE_MMS	D		
			RECEIVE_SMS	D		
			RECEIVE_WAP_PUSH	D		
			RECORD_AUDIO	D		
			REORDER_TASKS	N		
			RESTART_PACKAGES	D		
			SEND_SMS	D		
			SET_ALWAYS_FINISH	–		
			SET_ANIMATION_SCALE	–		
			SET_DEBUG_APP	–		
			SET_PREFERRED_APPLICATIONS	D		
			SET_PROCESS_LIMIT	–		

Table 3 continued

API	Date	OS version	Added permission	Type	Deprecated permission	Set
			SET_TIME_ZONE	–		
			SET_WALLPAPER	N		
			SET_WALLPAPER_HINTS	N		
			SIGNAL_PERSISTENT_PROCESSES	–		
			STATUS_BAR	–		
			SYSTEM_ALERT_WINDOW	O		
			VIBRATE	N		
			WAKE_LOCK	N		
			WRITE_APN_SETTINGS	–		
			WRITE_CALENDAR	D		
			WRITE_CONTACTS	D		
			WRITE_GSERVICES	–		
			WRITE_SETTINGS	O		
			WRITE_SYNC_SETTINGS	N		
2	02-2009	1.1 Petit Four	BROADCAST_SMS	–		76
			BROADCAST_WAP_PUSH	–		
3	04-2009	1.5 Cupcake	BIND_APPWIDGET	–		81
			BIND_INPUT_METHOD	O		
			MOUNT_FORMAT_FILESYSTEMS	–		
			UPDATE_DEVICE_STATS	–		
			WRITE_SECURE_SETTINGS	–		
4	09-2009	1.6 Donut	CHANGE_WIFI_MULTICAST_STATE	N		85
			GLOBAL_SEARCH	O		
			INSTALL_LOCATION_PROVIDER	–		
			WRITE_EXTERNAL_STORAGE	D		
5	10-2009	2.0 Eclair	ACCOUNT_MANAGER	–		86
6	12-2009	2.0.1 Eclair				86
7	01-2010	2.1 Eclair				86
8	05-2010	2.2 Froyo	BIND_DEVICE_ADMIN	O		90
			BIND_WALLPAPER	O		
			KILL_BACKGROUND_PROCESSES	N		
			SET_TIME	–		

Table 3 continued

API	Date	OS version	Added permission	Type	Deprecated permission	Set
9	12-2010	2.3 Gingerbread	NFC	N		93
			SET_ALARM	N		
			USE_SIP	D		
10	02-2011	2.3.3 Gingerbread				93
11	02-2011	3.0 Honeycomb	BIND_REMOTEVIEWS	O		94
12	05-2011	3.1 Honeycomb				94
13	07-2011	3.2 Honeycomb				94
14	10-2011	4.0 Ice Cream Sandwich	ADD_VOICEMAIL	D		97
			BIND_TEXT_SERVICE	O		
			BIND_VPN_SERVICE	O		
15	12-2011	4.0.3 Ice Cream Sandwich			PERSISTENT_ACTIVITY (1)	94
					RESTART_PACKAGES (1)	
					SET_PREFERRED_APPLICATIONS (1)	
16	07-2012	4.1 Jelly Bean	BIND_ACCESSIBILITY_SERVICE	O		97
		READ_CALL_LOG	D			
		READ_EXTERNAL_STORAGE	D			
		WRITE_CALL_LOG	D			
				READ_INPUT_STATE (1)		
17	11-2012	4.2 Jelly Bean				97
18	07-2013	4.3 Jelly Bean	BIND_NOTIFICATION_LISTENER_SERVICE	O		100
			LOCATION_HARDWARE	–		
			SEND_RESPOND_VIA_MESSAGE	–		
19	10-2013	4.4 KitKat	BIND_NFC_SERVICE	O		109
			BIND_PRINT_SERVICE	O		
			BLUETOOTH_PRIVILEGED	–		
			CAPTURE_AUDIO_OUTPUT	–		
			INSTALL_SHORTCUT	N		
			MANAGE_DOCUMENTS	–		
			MEDIA_CONTENT_CONTROL	–		
			TRANSMIT_IR	N		
			UNINSTALL_SHORTCUT	–		
20	06-2014	4.4W KitKat	BODY_SENSORS	D		110

Table 3 continued

API	Date	OS version	Added permission	Type	Deprecated permission	Set
21	11-2014	5.0 Lollipop	BIND_DREAM_SERVICE	O		114
			BIND_TV_INPUT	O		
			BIND_VOICE_INTERACTION	O		
			READ_VOICEMAIL	O		
			WRITE_VOICEMAIL	O		
					GET_TASKS (1)	
22	03-2015	5.1 Lollipop	BIND_CARRIER_MESSAGING_SERVICE	O		115
23	10-2015	6.0 Marshmallow	ACCESS_NOTIFICATION_POLICY	N		125
			BIND_CARRIER_SERVICES	O		
			BIND_CHOOSER_TARGET_SERVICE	O		
			BIND_INCALL_SERVICE	O		
			BIND_MIDI_DEVICE_SERVICE	O		
			BIND_TELECOM_CONNECTION_SERVICE	O		
			GET_ACCOUNTS_PRIVILEGED	O		
			PACKAGE_USAGE_STATS	O		
			REQUEST_IGNORE_BATTERY_OPTIMIZATIONS	N		
			REQUEST_INSTALL_PACKAGES	O		
			USE_FINGERPRINT	N		
					BIND_CARRIER_MESSAGING_SERVICE (22)	
24	08-2016	7.0 Nougat	BIND_CONDITION_PROVIDER_SERVICE	O		129
			BIND_QUICK_SETTINGS_TILE	–		
			BIND_SCREENING_SERVICE	O		
			BIND_VR_LISTENER_SERVICE	O		
25	10-2016	7.1 Nougat				129
26	08-2017	8.0 Oreo	ANSWER_PHONE_CALLS	D		138
			BIND_AUTOFILL_SERVICE	O		
			BIND_VISUAL_VOICEMAIL_SERVICE	O		
			INSTANT_APP_FOREGROUND_SERVICE	O		
			MANAGE_OWN_CALLS	N		
			READ_PHONE_NUMBERS	D		
			REQUEST_COMPANION_RUN_IN_BACKGROUND	N		
			REQUEST_COMPANION_USE_DATA_IN_BACKGROUND	N		
			REQUEST_DELETE_PACKAGES	N		
27	12-2017	8.1 Oreo				138

Table 3 continued

API	Date	OS version	Added permission	Type	Deprecated permission	Set
28	08-2018	9.0 Pie	ACCEPT_HANDOVER	D		141
			FOREGROUND_SERVICE	N		
			NFC_TRANSACTION_EVENT	N		
			USE_BIOMETRIC	N		
					USE_FINGERPRINT (23)	
29	09-2019	10 Quince Tart	ACCESS_BACKGROUND_LOCATION	D		150
			ACCESS_MEDIA_LOCATION	D		
			ACTIVITY_RECOGNITION	D		
			BIND_CALL_REDIRECTION_SERVICE	O		
			BIND_CARRIER_MESSAGING_CLIENT_SERVICE	O		
			CALL_COMPANION_APP	N		
			REQUEST_PASSWORD_COMPLEXITY	N		
			SMS_FINANCIAL_TRANSACTIONS	O		
			START_VIEW_PERMISSION_USAGE	O		
			USE_FULL_SCREEN_INTENT	N		
					PROCESS_OUTGOING_CALLS (1)	
30	09-2020	11 Red Velvet Cake	BIND_CONTROLS	–		157
			BIND_QUICK_ACCESS_WALLET_SERVICE	O		
			INTERACT_ACROSS_PROFILES	O		
			LOADER_USAGE_STATS	O		
			MANAGE_EXTERNAL_STORAGE	O		
			NFC_PREFERRED_PAYMENT_INFO	N		
			QUERY_ALL_PACKAGES	N		
			READ_PRECISE_PHONE_STATE	O		
					BIND_CHOOSER_TARGET_SERVICE (23)	

References

1. Hautala, L.: Android malware tries to trick you. Here's how to spot it. https://www.cnet.com/tech/services-and-software/android-malware-tries-to-trick-you-heres-how-to-spot-it/ (2021)
2. Palmer, D.: Sophisticated android malware spies on smartphones users and runs up their phone bill too. https://www.zdnet.com/article/sophisticated-android-malware-spies-on-smartphones-users-and-runs-up-their-phone-bill-too/ (2018)
3. Yaswant, A.: New advanced android malware posing as "ystem update". https://blog.zimperium.com/new-advanced-android-malware-posing-as-system-update/ (2021)
4. O'Dea, S.: Mobile operating systems' market share worldwide from January 2012 to June 2021. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (2021)
5. Kaspersky: Can you get viruses on android? every android user is at risk. https://www.kaspersky.com/resource-center/preemptive-safety/android-malware-risk (2021)
6. Velzian, B.: Calling all threat hunters—mobile malware to look out for in 2021. https://www.wandera.com/calling-all-threat-hunters-mobile-malware-to-look-out-for-in-2021/ (2021)
7. Android: App permissions best practices. https://developer.android.com/training/permissions/usage-notes (2021)
8. Google: Google play protect. https://developers.google.com/android/play-protect (2021)
9. Samsung: This is protection, samsung knox. https://www.samsungknox.com/en/secured-by-knox (2021)
10. Withwam, R.: Android antivirus apps are useless—here's what to do instead. https://www.extremetech.com/computing/104827-android-antivirus-apps-are-useless-heres-what-to-do-instead (2020)
11. Lakshmanan, R.: Joker malware apps once again bypass Google's security to spread via play store. https://thehackernews.com/2020/07/joker-android-mobile-virus.html (2020)
12. Chebyshev, V.: Mobile malware evolution 2020. https://securelist.com/mobile-malware-evolution-2020/101029 (2021)
13. Faruki, P., Ganmoor, V., Laxmi, V., Gaur, M.S., Bharmal, A.: Androsimilar: robust statistical feature signature for android malware detection. In: Proceedings of the 6th International Conference on Security of Information and Networks, pp. 152–159 (2013)
14. Feizollah, A., Anuar, N.B., Salleh, R., Wahab, A.W.A.: A review on feature selection in mobile malware detection. Digit. Investig. **13**, 22–37 (2015)
15. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: effective and explainable detection of android malware in your pocket. In: Ndss, vol. 14, pp. 23–26 (2014)
16. Lipovský, R., Štefanko, L., Braniša, G.: The rise of android ransomware. https://www.welivesecurity.com/wp-content/uploads/2016/02/Rise_of_Android_Ransomware.pdf (2016)
17. Mathur, A., Podila, L.M., Kulkarni, K., Niyaz, Q., Javaid, A.Y.: Naticusdroid: a malware detection framework for android using native and custom permissions. J. Inf. Secur. Appl. **58**, 102696 (2021)
18. Khariwal, K., Singh, J., Arora, A.: Ipdroid: android malware detection using intents and permissions. In: 2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4), pp. 197–202. IEEE (2020)
19. Android: Request app permissions. https://developer.android.com/training/permissions/requesting (2021)
20. Android: Permissions on android. https://developer.android.com/guide/topics/permissions/overview (2021)
21. Android: Manifest.permission., https://developer.android.com/reference/android/Manifest.permission (2021)
22. Android: Define a custom app permission. https://developer.android.com/guide/topics/permissions/defining (2021)
23. Codepath: Understanding app permissions. https://guides.codepath.com/android/Understanding-App-Permissions (2021)
24. Android: App permissions best practices. https://developer.android.com/training/permissions/usage-notes (2021)
25. Android: Permissions updates in android 11. https://developer.android.com/about/versions/11/privacy/permissions (2021)
26. JR, R.: Android versions: A living history from 1.0 to 12. https://www.computerworld.com/article/3235946/android-versions-a-living-history-from-1-0-to-today.html (2021)
27. Milosevic, N., Dehghantanha, A., Choo, K.-K.R.: Machine learning aided android malware classification. Comput. Electr. Eng. **61**, 266–274 (2017)
28. Zhu, H.-J., You, Z.-H., Zhu, Z.-X., Shi, W.-L., Chen, X., Cheng, L.: Droiddet: effective and robust detection of android malware using static analysis along with rotation forest model. Neurocomputing **272**, 638–646 (2018)
29. Talha, K.A., Alper, D.I., Aydin, C.: Apk auditor: permission-based android malware detection system. Digit. Investig. **13**, 1–14 (2015)
30. Rovelli, P., Vigfússon, Ý.: Pmds: permission-based malware detection system. In: International Conference on Information Systems Security. Springer, pp. 338–357 (2014)
31. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P.G., Álvarez, G.: Puma: permission usage to detect malware in android. In: International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions, pp. 289–298. Springer (2013)
32. Zarni Aung, W.Z.: Permission-based android malware detection. Int. J. Sci. Technol. Res. **2**, 228–234 (2013)
33. Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., Zhang, X.: Exploring permission-induced risk in android applications for malicious application detection. IEEE Trans. Inf. Forensics Secur. **9**, 1869–1882 (2014)
34. Ghasempour, A., Sani, N.F.M., Abari, O.J.: Permission extraction framework for android malware detection. Int. J. Adv. Comput. Sci. Appl. **11**(11) (2020)
35. Arora, A., Peddoju, S.K., Conti, M.: Permpair: android malware detection using permission pairs. IEEE Trans. Inf. Forensics Secur. **15**, 1968–1982 (2019)
36. Liu, X., Liu, J.: A two-layered permission-based android malware detection scheme. In: 2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering, pp. 142–148 (2014). https://doi.org/10.1109/MobileCloud.2014.22
37. Moonsamy, V., Rong, J., Liu, S.: Mining permission patterns for contrasting clean and malicious android applications. Futur. Gener. Comput. Syst. **36**, 122–132 (2014)
38. Sokolova, K., Perez, C., Lemercier, M.: Android application classification and anomaly detection with graph-based permission patterns. Decis. Support Syst. **93**, 62–76 (2017)
39. Wang, C., Xu, Q., Lin, X., Liu, S.: Research on data mining of permissions mode for android malware detection. Clust. Comput. **22**, 13337–13350 (2019)
40. Idrees, F., Rajarajan, M., Conti, M., Chen, T.M., Rahulamathavan, Y.: Pindroid: a novel android malware detection system using ensemble learning methods. Comput. Secur. **68**, 36–46 (2017)
41. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Nieves, J., Bringas, P.G., Álvarez Marañón, G.: Mama: manifest analysis for malware detection in android. Cybern. Syst. **44**, 469–488 (2013)
42. Arslan, R.S., Ölmez, E., Er, O.: Afwdroid: deep feature extraction and weighting for android malware detection. Dicle üniversitesi MÜhendislik Fakültesi Mühendislik Dergisi **12**, 237–245 (2021)
43. Alazab, M., Alazab, M., Shalaginov, A., Mesleh, A., Awajan, A.: Intelligent mobile malware detection using permission requests and API calls. Futur. Gener. Comput. Syst. **107**, 509–521 (2020)

SPI Journal: 11416 MS: 0432 TYPESET DISK LE CP Disp.:2022/8/22 Pages: 32 Layout: Large

44. Tao, G., Zheng, Z., Guo, Z., Lyu, M.R.: Malpat: mining patterns of malicious and benign android apps via permission-related APIs. IEEE Trans. Reliab. **67**, 355–369 (2017)
45. Kim, T., Kang, B., Rho, M., Sezer, S., Im, E.G.: A multimodal deep learning method for android malware detection using various features. IEEE Trans. Inf. Forensics Secur. **14**, 773–788 (2018)
46. Hu, D., Ma, Z., Zhang, X., Li, P., Ye, D., Ling, B.: The concept drift problem in android malware detection and its solution. Secur. Commun. Netw. **2017** (2017)
47. Guerra-Manzanares, A., Nomm, S., Bahsi, H.: In-depth feature selection and ranking for automated detection of mobile malware. In: ICISSP, pp. 274–283 (2019)
48. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. In: NDSS, vol. 25, pp. 50–52 (2012)
49. Lindorfer, M., Neugschwandtner, M., Platzer, C.: Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In: IEEE 39th Annual Computer Software and Applications Conference, vol. 2, pp. 422–433. IEEE (2015)
50. Arora, A., Peddoju, S.K.: Ntpdroid: a hybrid android malware detector using network traffic and system permissions. In: 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE), pp. 808–813. IEEE (2018)
51. Arora, A., Peddoju, S.K., Chouhan, V., Chaudhary, A.: Hybrid android malware detection by combining supervised and unsupervised learning. In: Proceedings of the 24th Annual International Conference on Mobile Computing and Networking, pp. 798–800 (2018)
52. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: IEEE Symposium on Security and Privacy, pp. 95–109. IEEE (2012)
53. Guerra-Manzanares, A., Bahsi, H., Nõmm, S.: Kronodroid: time-based hybrid-featured dataset for effective android malware detection and characterization. Comput. Secur. **110**, 102399 (2021)
54. Mila: Contagio mobile. http://contagiominidump.blogspot.com/ (2018)
55. Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., Rieck, K.: Dos and don'ts of machine learning in computer security (2020). arXiv preprint arXiv:2010.09470
56. Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L.: {TESSERACT}: eliminating experimental bias in malware classification across space and time. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 729–746 (2019)
57. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Are your training datasets yet relevant? In: International Symposium on Engineering Secure Software and Systems, pp. 51–67. Springer (2015)
58. Cen, L., Gates, C.S., Si, L., Li, N.: A probabilistic discriminative model for android malware detection with decompiled source code. IEEE Trans. Dependable Secure Comput. **12**, 400–412 (2015)
59. Xu, K., Li, Y., Deng, R., Chen, K., Xu, J.: Droidevolver: self-evolving android malware detection system. In: IEEE European Symposium on Security and Privacy (EuroS&P), pp. 47–62. IEEE (2019)
60. Lei, T., Qin, Z., Wang, Z., Li, Q., Ye, D.: Evedroid: event-aware android malware detection against model degrading for ioT devices. IEEE Internet Things J. **6**, 6668–6680 (2019)
61. Guerra-Manzanares, A., Luckner, M., Bahsi, H.: Android malware concept drift using system calls: detection, characterization and challenges. Expert Syst. Appl. **117200**, 117200 (2022)
62. Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., Zhang, G.: Learning under concept drift: a review. IEEE Trans. Knowl. Data Eng. **31**, 2346–2363 (2018)
63. Lu, N., Zhang, G., Lu, J.: Concept drift detection via competence models. Artif. Intell. **209**, 11–28 (2014)
64. Jordaney, R., Sharad, K., Dash, S.K., Wang, Z., Papini, D., Nouretdinov, I., Cavallaro, L.: TranscEnd: detecting concept drift in malware classification models. In: Proceedings of the 26th USENIX Security Symposium, pp. 625–642 (2017)
65. Hooker, G., Mentch, L.: Please stop permuting features: an explanation and alternatives (2019). arXiv preprint arXiv:1905.03151
66. Samara, B., Randles, R.H.: A test for correlation based on Kendall's tau. Commun. Stat. Theory Methods **17**, 3191–3205 (1988)
67. Aggarwal, C.C.: Data Mining: The Textbook. Springer, Berlin (2015)
68. Zyblewski, P., Sabourin, R., Woźniak, M.: Preprocessed dynamic classifier ensemble selection for highly imbalanced drifted data streams. Inf. Fusion **66**, 138–154 (2021)
69. Guerra-Manzanares, A., Nõmm, S., Bahsi, H.: Time-frame analysis of system calls behavior in machine learning-based mobile malware detection. In: 2019 International Conference on Cyber Security for Emerging Technologies (CSET), pp. 1–8 (2019). https://doi.org/10.1109/CSET.2019.8904908
70. Guerra-Manzanares, A., Bahsi, H., Nõmm, S.: Differences in android behavior between real device and emulator: a malware detection perspective. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), pp. 399–404 (2019). https://doi.org/10.1109/IOTSMS48152.2019.8939268
71. Maimon, O., Rokach, L. (eds.): Data Mining and Knowledge Discovery Handbook. A Complete Guide for Practitioners and Researchers. Springer, San Francisco (2005)
72. Breiman, L.: Random forests. Mach. Learn. **45**, 5–32 (2001)
73. Altmann, A., Toloşi, L., Sander, O., Lengauer, T.: Permutation importance: a corrected feature importance measure. Bioinformatics **26**, 1340–1347 (2010)
74. Biecek, P., Burzykowski, T.: Explanatory Model Analysis. Chapman and Hall, New York (2021)
75. Molnar, C.: Interpretable machine learning. https://christophm.github.io/interpretable-ml-book/ (2019)
76. Shapley, L.S.: 17. A Value for n-person Games. Princeton University Press, Princeton (2016)
77. Japkowicz, N., Shah, M.: Evaluating Learning Algorithms: A Classification Perspective. Cambridge University Press, New York (2011)
78. Breunig, M.M., Kriegel, H.-P., Ng, R.T., Sander, J.: Lof: identifying density-based local outliers. In: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, pp. 93–104 (2000)
79. Wu, L.: Android mobile ransomware: bigger, badder, better? https://www.trendmicro.com/en_us/research/17/h/android-mobile-ransomware-evolution.html (2017)
80. Seals, T.: Slocker android ransomware resurfaces in undetectable form. https://www.infosecurity-magazine.com/news/slocker-android-ransomware/ (2017)

SPI Journal: **11416** MS: **0432** ☐ TYPESET ☐ DISK ☐ LE ☐ CP Disp.:**2022/8/22** Pages: **32** Layout: **Large**

Appendix 9

Publication IX
A. Guerra-Manzanares and H. Bahsi. On the relativity of time: Implications and challenges of data drift on long-term effective android malware detection. *Computers & Security*, *in press*:102835, 2022

Computers & Security 122 (2022) 102835

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

On the relativity of time: Implications and challenges of data drift on long-term effective android malware detection

Alejandro Guerra-Manzanares*, Hayretdin Bahsi

Department of Software Science, Tallinn University of Technology, Estonia

ARTICLE INFO

Article history:
Received 24 March 2022
Revised 29 June 2022
Accepted 12 July 2022
Available online 16 July 2022

Keywords:
Android malware
Machine learning
Timestamp
Malware detection
Concept drift
Malware evolution
Data drift

ABSTRACT

The vast body of research in the Android malware detection domain has demonstrated that machine learning can provide high performance for mobile malware detection. However, the learning models have been usually evaluated with data sets encompassing short time frames, generating doubts about the feasibility of these models in operational settings that deal with the ever-evolving malware threat landscape. Although a limited number of studies have developed concept drift resilient models for handling data drift, they have never considered the impact of different timestamps on the detection solutions. Timestamps are critical to locating the data samples within the historical timeline. Different timestamping approaches may locate samples differently, which, in turn, can significantly impact the performance of the model and, consequently, the adaptive capabilities of the system to concept drift. In this study, we conducted a comprehensive benchmarking that compares the detection performance of six distinct timestamping approaches for static and dynamic feature sets. Our experiments have demonstrated that timestamp selection is an important decision that has a significant impact on concept drift modeling and the long-term performance of the model regardless of the feature type used for model construction.

1. Introduction

Mobile devices have continuously penetrated every aspect of our lives, including personal and business-related. The vibrant ecosystem of mobile app development has accompanied the progress in the areas of mobile hardware and OS, leading to the very rapid uptake of technology. Considering the increasing involvement of mobile apps in the management of IoT devices, mobile security threats may cause economic, societal, or even physical damage. In this threat category, malicious applications pose a significant attack vector due to their evolving nature. Despite the countermeasures applied by device and OS vendors (Google, 2021; Samsung, 2021), the threat still remains. Therefore, it is imperative to timely detect and isolate the apps showing malicious behavior. This study focuses on Android malware, as Android is the dominant OS in the market (72% as of September 2021 (Statista, 2021)) and the target of most mobile malware (Islam, 2021).

Machine learning (ML) methods have provided promising detection results in Android malware detection (Sharma and Rattan, 2021). However, the general validation approach in these studies is limited to the utilization of data sets that encompass specific time frames without paying attention to the detection performance against the evolving threat landscape that includes new malware types and variants of existing ones.

Here, it is important to underline the general expectation of the cyber security domain about the ML methods in detecting malicious behavior. The current signature-based solutions supported by the cyber threat intelligence ecosystem constitute a working solution despite its various well-known shortcomings. ML methods extend this capability by detecting malware not seen before (Buczak and Guven, 2015). However, *temporal experimental bias* arises as a significant issue when the models are not tested with samples that belong to later times than the training samples (Pendlebury et al., 2019). The existence of such bias creates confusion about whether the model can detect new and evolved malware and whether the operational environments can sustain their discriminatory capability for a long time against data shifts in malware or benign data. Therefore, it is necessary to model the change in the threat landscape (i.e., concept drift) by considering longer time frames and determining the life cycle of the detection models (e.g., creation and update). We even argue that handling the concept drift should be an integral part of any ML-based malware detection solution to demonstrate its viability in operational settings.

Regardless of the utilized features or modeling approaches, timestamps are the central element for concept drift analysis and

* Corresponding author.
E-mail addresses: alejandro.guerra@taltech.ee (A. Guerra-Manzanares), hayretdin.bahsi@taltech.ee (H. Bahsi).

https://doi.org/10.1016/j.cose.2022.102835
0167-4048/

its proper handling as they provide the grounds of the *historical* context. They enable us to locate every app within the Android historical timeline according to their values. In this regard, the misplacement of a large number of samples may distort the data distribution and feature space representation. As a result, an inaccurate *timestamping* approach may prevent the model grasp the natural evolution of malware, which is vital for proper detection. However, despite the critical role of timestamps, there is no unambiguous approach to determining an app's temporal location with complete reliability and accuracy. A mobile app includes many files with temporal metadata that can be used as a timestamp for the app. External information sources such as *VirusTotal* can also provide the temporal context regarding the appearance of malware *in the wild*. It is hard to define the exact timing of the appearance of specific malware and malware families as they usually evolve from older variants or reuse code from other malware types.

Despite their importance, timestamping methods and their impact on learning models have not been thoroughly investigated. In the literature, only Guerra-Manzanares et al. (2022b) used two timestamps to locate samples in the system calls feature space analyzing cross-platform detection issues. In contrast, we conducted a benchmarking study that investigates the impact of six distinct temporal approaches on concept drift-based models induced by using a dynamic feature set, system calls, and two static feature sets, API calls and permissions. As a data set, we utilized *KronoDroid*, which covers a wide period (i.e., 2008–2020), in addition to providing various timestamp alternatives (Guerra-Manzanares et al., 2021). First, we performed a comprehensive statistical analysis of malicious and benign apps to explore the *availability* and *validity* of the temporal data obtained by each timestamping approach. In this part, we also developed an approximation method to compare the accuracy of the obtained data. Then, we formulated a concept drift-handling method that uses a *pool* composed of classifiers to compare the detection rates of each timestamping approach for each feature set.

Our results indicate that the selection of the timestamping method has a significant impact on the detection accuracy of the learning models that encompass long time frames regardless of the feature set nature (i.e., dynamic or static). Thus, it is imperative to consider the collection of relevant temporal values besides the comprehensiveness of the data set. Note that our work does not aim to optimize the performance of the concept drift-handling method. Instead, it uses a working solution to compare the performance of temporal data for each feature set.

Our study provides a unique contribution to the literature as a detailed evaluation regarding the impact of timestamping alternatives on the performance of concept drift models has not been addressed for mobile malware detection.

The outline of the paper is as follows: Section 2 defines concept drift and its consequences, while Section 3 surveys the related literature. Methods and analytical approaches used in the study are presented in Section 4. Section 5 provides the results of our experiments. The key findings, contributions and limitations of our study are given in Section 6. Section 7 concludes the paper.

2. Concept drift

Most learning models are *static*, meaning that they assume stationary data distributions, which are consistent over time. Thus, the training and testing data are assumed to be *similar*. However, non-stationary data distributions might be found in many problem domains, such as in Android malware detection, where data evolves over time (e.g., the emergence of new malware families), adding additional complexity to the data modeling process that, if not addressed, can severely impact the generalization of the detection model to future (evolved) data leading to model obsolescence, a phenomenon called *concept drift*.

Concept drift can be defined as the phenomenon in which the statistical properties of data change over time in an unpredictable manner (Lu et al., 2018). Formally, concept drift can be defined as follows.

Given a bounded period of time $[t_0, t_1]$ and a set of examples from that period $S_{t_0.t_1} = \{s_{t_0}, \ldots, s_{t_1}\}$, where $s_i = (x_i, y_i)$ relates to a single observation, $x_i = (x_i^1, x_i^2, \ldots, x_i^n) \in \mathbf{X}$ defines the feature vector, $y_i \in \mathbf{Y}$ corresponds to the target label, and $S_{t_0.t_1}$ follows a specific distribution $F_{t_0.t_1}(\mathbf{X}, \mathbf{Y})$. The phenomenon of concept drift is observed at t_2 if $F_{t_0.t_1}(\mathbf{X}, \mathbf{Y}) \neq F_{t_2.\infty}(\mathbf{X}, \mathbf{Y})$, and may be denoted as $\exists t : P_t(\mathbf{X}, \mathbf{Y}) \neq P_{t+1}(\mathbf{X}, \mathbf{Y})$ (Lu et al., 2018). Following this definition, concept drift at time period t_i can be equated as the change in the joint probability of $\mathbf{X}$ and $\mathbf{Y}$ at time t_i, expressed as $P_{t_i}(\mathbf{X}, \mathbf{Y})$. As $P_{t_i}(\mathbf{X}, \mathbf{Y}) = P_{t_i}(\mathbf{X}) \times P_{t_i}(\mathbf{Y} \mid \mathbf{X})$, concept drift can arise from three primary sources (Lu et al., 2018):

1) $P_t(\mathbf{X}) \neq P_{t+1}(\mathbf{X})$ and $P_t(\mathbf{Y} \mid \mathbf{X}) = P_{t+1}(\mathbf{Y} \mid \mathbf{X})$. This shows a change in the input data distribution, $P_t(\mathbf{X})$, that has no impact on the posterior probability of the learning model, $P_t(\mathbf{Y} \mid \mathbf{X})$, thus not affecting its decision boundary. This phenomenon is named *virtual drift*.
2) $P_t(\mathbf{X}) = P_{t+1}(\mathbf{X})$ and $P_t(\mathbf{Y} \mid \mathbf{X}) \neq P_{t+1}(\mathbf{Y} \mid \mathbf{X})$. In such conditions, the input data distribution does not change, but the changes in the posterior probability modify the decision boundary, leading to a decrease in the learning accuracy, which reflects *real concept drift*.
3) $P_t(\mathbf{X}) \neq P_{t+1}(\mathbf{X})$ and $P_t(\mathbf{Y} \mid \mathbf{X}) \neq P_{t+1}(\mathbf{Y} \mid \mathbf{X})$. In such a scenario, the shift in the feature data distribution coincides with a change in the decision boundary, defining a *real concept drift*.

As can be noted from the previous definitions, only the *real* concept drift changes the decision boundary of the model, which affects the generalization capabilities of the model and leads to model obsolescence. More precisely, *real concept drift* relates to changes in the model's decision boundary, defined by $P(\mathbf{Y} \mid \mathbf{X})$, that might coexist with *covariate shift* (Gama et al., 2014). *Virtual drift*, also named *feature space drift* or *covariate shift*, reflects a change in the underlying data distribution that does not affect the model decision boundary, that is, $P(\mathbf{Y} \mid \mathbf{X})$. In this regard, from a predictive perspective, only the changes that affect the decision boundary, that is, the *class* prediction made by the model, require the implementation of adaptive measures (Gama et al., 2014).

3. Related work

3.1. Concept drift in android malware detection

The vast majority of literature regarding Android malware detection neglects the existence of concept drift. The models and proposed solutions are generally trained and validated on static *snapshots* of Android historical data, usually with the same *well-known* data sets (Abderrahmane et al., 2019; Afonso et al., 2015; Ahsan-Ul-Haque et al., 2018; Alzaylaee et al., 2017; 2020; Amin et al., 2016; Bhatia and Kaushal, 2017; Burguera et al., 2011; Canfora et al., 2015; Casolare et al., 2021; Da et al., 2016; Dimjašević et al., 2016; Feng et al., 2018; Ferrante et al., 2016; Frenklach et al., 2021; Guerra-Manzanares et al., 2019a; 2019b; 2019c; Hei et al., 2021; Hou et al., 2016; Isohara et al., 2011; Jaiswal et al., 2018; Jang et al., 2014; Kapratwar et al., 2017; Lin et al., 2013; Lindorfer et al., 2015; Liu et al., 2021; Mahindru and Sangal, 2021; Malik and Khatter, 2016; McLaughlin et al., 2017; Naval et al., 2015; Rathore et al., 2021; Saif et al., 2018; Saracino et al., 2018; Sasidharan and Thomas, 2021; Sihag et al., 2021; Singh and Hofmann, 2017; Surendran et al., 2020; Tchakounté and Dayang, 2013; Tong and Yan, 2017; Vidal et al., 2017; Vinod et al., 2019;

Wahanggara and Prayudi, 2015; Wang and Li, 2021; Xiao et al., 2015; 2016; 2019; Yu et al., 2013; Yuan et al., 2014). In this regard, *MalGenome* (Zhou and Jiang, 2012) and *Drebin* (Arp et al., 2014) are the most used data sets for Android malware research. Despite their small size and *outdated* data (i.e., they were collected between 2009–2012), they are still widely used as the primary source of malware samples in research studies (Rathore et al., 2021; Sasidharan and Thomas, 2021).

Even though some recent studies (Cai et al., 2021a; 2021b; Gao et al., 2021) complement their data with more recent and larger data sets, such as the *Android Malware Dataset* (AMD) (Wei et al., 2017), to mitigate data-related issues (e.g., Drebin duplication issue (Irolla and Dey, 2018)), they still rely on *partial*, relatively *old* (e.g., AMD's most recent sample dates back to 2016 and includes just 71 malware families), and *short* snapshots of data within the whole Android historical timeline (i.e., 2008–2022). Furthermore, when the proposed solutions for Android malware detection are generated, it is common to randomize the data set order, thus neglecting its *natural* ordering, and split it into two disjoint sets (i.e., train/test sets) or several train/test folds (i.e., cross-validation). This common procedure, which can be applied to any ML task where the data does not follow any natural order or evolution issues (e.g., image classification), can significantly harm the performance of solutions built for *naturally* ordered sequential data, such as natural language processing tasks or data streams. The randomization of train/test splits neglects apps' location in the historical timeline, undermining *historical coherence* and yielding *significantly biased* and *historically incoherent* results (Allix et al., 2015; Pendlebury et al., 2019). As a result, the training data may contain samples belonging to *future* time frames, and the testing data may include *past* samples contemporaneous to the training data. Thus the model is trained with an *impossible configuration*, with data that is not typically available in practice and providing biased performance, a phenomenon called *data snooping* (Arp et al., 2020). Consequently, these practices pose serious doubts about the *generalization* capabilities and effectiveness of these solutions for detecting evolved and recent malware.

Only a reduced number of the referred studies considered the usage of distinct snapshots of historical data for the train/test split. However, they included significant time gaps between them (Guerra-Manzanares et al., 2019a; 2019b; 2019c). Therefore, *concept drift* and its degenerative impact on the performance of the ML-based models were neglected.

As a result, the *time* variable and malware evolution have been neglected by the vast majority of Android malware research. Only a few studies dealing with Android malware detection have considered the concept drift issue and proposed ML solutions that *adapt* to changes in the data and aim to minimize its detrimental impact over time. In this regard, general frameworks have been proposed to detect emerging *drift* in Android data (Barbero et al., 2020; Jordaney et al., 2017; Pendlebury et al., 2019), although the vast majority of proposed solutions are based on API calls (Cai, 2020; Lei et al., 2019; Narayanan et al., 2016; Onwuzurike et al., 2019; Xu et al., 2019; Zhang et al., 2020).

3.2. Timestamps: when time matters

The essential constructs underlying effective concept drift handling are *timestamps*. Timestamps enable the temporal allocation of apps, aiming to provide a reliable temporal context. However, due to the nature of the malware generation and discovery process, the reliability of timestamps is questionable.

The studies that tackled concept drift-related issues in Android malware detection have not discussed the issue of timestamp selection. Although some studies did not provide any information about it (Onwuzurike et al., 2019), most studies used a similar approach, the *compilation date*. The compilation date, referred to using different names in the literature, is an *internal* timestamp that reports the creation or compilation time of the app archive (i.e., *apk*). Despite being pointed out as the best timestamp (Pendlebury et al., 2019) and used in related research (Barbero et al., 2020; Cai et al., 2020; Pendlebury et al., 2019; Xu et al., 2019), it has become an unusable approach as most of the apps released nowadays have it set in *1980* (du Luxembourg, 2021). Another *internal* timestamp, proposed more recently, is the last modification timestamp, which refers to the most recent modification *datetime* found in any of the inner files of the *apk* archive (Guerra-Manzanares et al., 2021). This feature was introduced in Guerra-Manzanares et al. (2021), which discussed the feasibility of four distinct timestamp approaches for Android malware detection.

Even though the internal timestamps, collected from apps' inner files, could be deemed *accurate*, they are prone to manipulation, which may cause invalid timestamps or temporal misplacement. A more robust temporal context can be obtained using *external* timestamps. An external timestamp is not retrieved from the app files but provided by an external entity. In this regard, Virustotal's *first seen*, also named *appearance* or *submission* time in the studies, reports the *datetime* at which a sample was first received by the VirusTotal scanning service. It has been used in research studies (Cai et al., 2019; Lei et al., 2019; Zhang et al., 2020) as it is based on third-party, reliable services, making the temporal assignment more robust to alterations. However, due to its proactive nature, this timestamp is prone to significant delay and temporal displacement (i.e., a user must submit the file to generate the timestamp).

Consequently, the timestamp attribute and its selection emerge as critical elements to handle concept drift effectively for long-term Android malware detection. Despite its criticality, it has been neglected by the concept drift-related studies in the related literature. In this research, we address this significant research gap by thoroughly analyzing, comparing, and evaluating different timestamps for effective concept drift handling.

4. Methodology

4.1. Data set

The data set used in this study is *KronoDroid* (Guerra-Manzanares et al., 2021). This data set provides timestamped and labeled Android app samples covering the whole 2008–2020 period. Each data sample is described using dynamic and static features. More precisely, it is composed of 489 features, of which 289 are dynamic features and 200 static features. For this research, system calls and permissions features were used along with the provided timestamps (i.e., four timestamps), class labels, and other relevant metadata. As Kronodroid is composed of two data sets (i.e., according to the source of the dynamic features), in this study, the *real device dataset* was preferred due to its larger size for the same historical time frame (i.e., 78,137 samples). Furthermore, the selection of real device-collected data prevents that any behavioral differences could be influenced by malware *anti-sandbox* techniques. Therefore, the data set used in this research is composed of 41,382 malware samples belonging to 240 malware families and 36,755 benign apps, encompassing from 2008 to 2020.

To explore the impact of the time variable and the concept drift issue extensively, additional features were collected for this study. More specifically, for each data sample, Android API calls were statically collected using *Androguard* (Desnos et al., 2018) and two additional timestamps from the inner *apk* files (i.e., *dex date* and *manifest date*). The usage of all these data features enables us to observe and analyze the *time* impact on distinct feature spaces for the same set of data within the same problem do-

Table 1
Feature spaces.

Feature space	Dimensions	Type
System calls	288	Numeric
Permissions	166	Binary
API calls	53,523	Binary

main (i.e., system calls, permissions, and API calls for malware detection) from six different temporal perspectives (i.e., timestamps). The sample descriptors (i.e., features) are representative of widely used dynamic and static approaches for Android malware detection (Feizollah et al., 2015).

The feature sets used in this research, their dimensions, and data type are summarized in Table 1. The temporal perspectives used to date the apps within the Android historical timeline and their description are outlined in Table 2.

As can be observed in Table 1, three feature spaces are analyzed in this research for the same data set, providing complementary perspectives for the same phenomenon (i.e., concept drift). Furthermore, as the feature spaces are defined on varying dimensions and data types, they enable us to explore and analyze the issue from a wide perspective. In addition, the temporal dimensions segment and transform the feature spaces distinctively, thus providing extensive exploration of the studied phenomenon. In this study, six distinct timestamps are used and thoroughly compared. KronoDroid data set provides four possible timestamps per sample: *last modification, earliest modification, first seen,* and *first seen in the wild*. The nature of their temporal attribution is described in Table 2. Besides these timestamps, the *dex date* and *manifest date* timestamps were extracted in this research for all the *apk*s that compose the original data set. These timestamps were collected by inspection of two important app inner files: *classes.dex* and *AndroidManifest.xml*. The *dex date* timestamp, not included in the *Kronodroid* data set due to the inaccurate value reported for recent samples (Guerra-Manzanares et al., 2021; du Luxembourg, 2021), has been widely used in previous related research to locate apps within the Android timeline, thus being relevant for the comparison. The *dex date* timestamp is also referred to in the literature as *compilation* or *creation* time, referring to the *classes.dex* timestamp, which is essentially the compiled source code. The *manifest date* reports the timestamp of the *AndroidManifest.xml* or *app manifest* file. The addition of the *manifest date* as a temporal approach is based on the critical relevancy of this file for any Android app. In this regard, the *manifest* is the only compulsory file for any Android app as it provides the essential information about the app to the Android build tools, the operating system, and Google Play (Android, 2021a). Therefore, the relevancy of this critical file and its *omnipresent* characteristic are leveraged as the grounds for its inclusion in a complete analysis.

4.2. Workflow

4.2.1. Data collection

The initial stage of this research involved the collection of additional features to describe every sample in the selected data set.

API *external* methods were extracted from every apk file using *Androguard* (Desnos et al., 2018). As not all *external* methods are Android API calls, the results were filtered to select only the methods relative to the existing Android API families (Android, 2021b). These *Android Platform API* families include *android, dalvik, java, javax, junit, apache, json, dom*, and *xml*. A similar approach was used in API call-related studies (Onwuzurike et al., 2019; Xu et al., 2019). Each class method defined by the apps in the data set was included as a feature. Therefore, all samples within the data set were described using a binary vector that indicates the presence/absence (i.e., 1/0) of each API call for each data sample.

The additional timestamps used to describe every data sample (i.e., *dex date* and *manifest date*) were retrieved from the inner files of the *apk* archive by inspecting the metadata extracted from the *classes.dex* and *AndroidManifest.xml* files. A *Python* script was used for that purpose.

The collected data were processed and structured in *CSV* format, where each row referred to a data sample and each column to a feature. As a result, each sample was described using a vector concatenating the data for the three independent feature spaces (i.e., syscalls, permissions, and API calls), the six timestamps, the class label, malware family, and the *sha-256* hash value that uniquely identifies each data sample.

The feature spaces explored in this research are representative of the most common attributes used for Android malware detection purposes. *System calls* are the most widely used dynamic features, whereas *permissions* and *API calls* are the features used in the vast majority of *static* approaches for malware detection (Feizollah et al., 2015).

4.2.2. Timestamp analysis

The main focus of this research is the temporal dimension of the data and its impact on concept drift representation, analysis, and handling. Therefore, in this second stage, a thorough comparative analysis of the timestamps was performed.

The usage of a timestamp to locate applications across the Android historical timeline is subject to *availability* and *reliability* issues. The former refers to the *accessibility* of the timestamp for its collection, while the latter regards the temporal precision of the timestamp concerning the ground-truth location of the app within the historical timeline. As the ground-truth temporal location is hardly achievable in the vast majority of cases (i.e., it is not possible to know with absolute certainty when the sample was released), the timestamp approaches main aim is to provide a *good approximation* to the given phenomenon in the absence of ground-truth data. In our approach, a good approximation would minimize the amount of error for the majority of the samples and enable the handling of concept *drifts* effectively. In this regard, due to the absence of a ground-truth temporal reference, our assumption is that an effective concept drift-handling solution may provide relatively more stable and smoother performance over time using an accurate timestamp than with an inaccurate timestamp, as data should usually evolve through shifting gradually over time, introducing new elements and discarding others in a relatively smooth transition. Sudden data drifts may happen over time, but their prevalence should not be significant (e.g., completely new malware outbreaks). Otherwise, concept drift could hardly be modeled, and keeping high performance over time would be a utopia.

At this stage, the following steps were performed to analyze and compare the timestamps using their intrinsic properties, whereas the next stage, described in Section 4.2.3, explores their suitability for concept drift handling.

The sequential steps of the comparative analysis are described as follows:

1) *Prevalence*: the prevalence of timestamps is a term related to *availability* that informs about the accessibility of the timestamp, that is, if the timestamp can be successfully collected or retrieved from the samples. For each timestamp, the number of set timestamps (i.e., properly defined) and not set timestamps (i.e., *missing* or undefined) was analyzed.
2) *Validity*: the validity of a timestamp is an indicator of whether the timestamp is comprised within the Android historical time frame (i.e., from the 22nd of October 2008 (Google, 2008) to the present day). It is not an indicator of accuracy but discards

Table 2
Timestamps.

Timestamp name	Description
Last modification	The most recent timestamp retrieved from any file inside the *apk* archive
Earliest modification	The *oldest* timestamp retrieved from any file inside the *apk* archive
First seen	Date and time of the first submission of the sample to *VirusTotal*
First seen in the wild	Date and time of the first time the sample was seen anywhere on internet
Dex date	Timestamp retrieved from the *classes.dex* file, located inside the *apk* archive (i.e., compilation time)
Manifest date	Timestamp retrieved from the *AndroidManifest.xml* file, located inside the *apk* archive

all the timestamps not comprised within the *valid* Android timeline range (i.e., before the 22nd of October 2008 or in the future).

3) *Suitability*: the suitability of a timestamp combines the previous concepts positively. Thus a *suitable* timestamp is *available* and *valid*. Consequently, an *unsuitable* timestamp is *available* but *invalid*.
4) *Distribution and statistical analysis*: data distributions for each timestamp and each class were analyzed and compared using statistical measures. Histograms were used to visualize the data distributions and as input to statistical tests and techniques for similarity assessment. Two statistical methods for measuring the similarity between data distributions were used:
   - *Jensen–Shannon distance*: a distance metric that is the square root of the *Jensen–Shannon divergence*. The Jensen–Shannon divergence is computed using the *Kullback Leibler (KL) divergence*, but it has more desirable properties than the KL divergence, such as being always finite, symmetrical, and smooth, thus preferred when some probability values are small or zero. The Jensen–Shannon divergence is computed as:

     $$JSD(P||Q) = \frac{D_{KL}(P||M)}{2} + \frac{D_{KL}(Q||M)}{2},$$

     where P and Q refer to two probability distributions, M is the point-wise mean of P and Q calculated as $\frac{1}{2}(P+Q)$ and D_{KL} refers to KL divergence values calculated for each pair of distributions. The KL divergence, also named *relative entropy*, between two discrete distributions is calculated as:

     $$D_{KL}(P||Q) = \sum_{x \in X} P(x) \log\left(\frac{P(x)}{Q(x)}\right)$$

     The KL divergence quantifies the difference between two probability distributions, which is leveraged by the Jensen–Shannon divergence to provide a more comprehensive, smooth, and well-defined distance metric (i.e., the square root of JSD) to measure the similarity between two probability distributions. The JSD distance for two probability distributions is bounded between [0, 1] when the logarithm to the base 2 is used for computations. The general interpretation is that the higher the value (i.e., closer to one), the greater the difference between the distributions.
   - *Kolmogorov–Smirnov two-sample test*: a non-parametric statistical hypothesis test to assess the equality of one-dimensional probability distributions. It enables us to assess the probability that two collections of samples (i.e., $F(x)$ and $G(x)$) could have been drawn from the same probability distribution, that is, if they are statistically similar. The null hypothesis H_0 for the test is that the two distributions are identical (i.e., $F(x) = G(x), \forall x$), whereas the alternative hypothesis H_1 is that they are not identical. The Kolmogorov-Smirnov (KS) test answers this hypothesis test by analyzing the maximum difference between the two experimental cumulative frequency distribution functions. The KS statistic is calculated as:

     $$D_{m,n} = \sup_x \mid F_n(x) - G_m(x) \mid,$$

     where $F_n(x)$ and $G_m(x)$ refer to the empirical distribution functions of the two data samples, of sizes m and n, respectively, and *sup* is the *supremum* function. For large samples, the null hypothesis is rejected at significance level α if

     $$D_{m,n} > c(\alpha)\sqrt{\frac{m+n}{mn}},$$

     where m and n are the sizes of the distributions and the value of $c(\alpha)$ is a parameter calculated as $c(\alpha) = \sqrt{-ln(\frac{\alpha}{2})\frac{1}{2}}$.

   Therefore, different combinations of data distributions based on the timestamps were analyzed regarding their similarity using both measures. As they evaluate *similarity* using different approaches, the usage of both techniques provides a better overall perspective of the differences between the analyzed sets.
5) *Accuracy*: an approximation of the accuracy of the timestamps was explored to assess their reliability. The evaluation of timestamp accuracy (i.e., the precise location of the sample within the Android historical timeline) is a significant challenge due to the absence of an exact ground-truth timestamp (i.e., it is not possible to surely know when the malware was first released). Thus only approximations to the ground-truth timestamp can be targeted. For this purpose, we used open-source intelligence feeds such as new malware family discovery news by antivirus vendors and media sources to establish an approximate *discovery time* of a specific malware family. After that, a time frame around the date was established (i.e., $\pm$ 6 months), and statistics were retrieved from each timestamp data distribution for each family. The rationale behind this analysis is that, if the timestamp is accurate, it should place samples around that time frame (i.e., *discovery time* $\pm$6 months) and after it, implying that the malware family might be located accurately and also its evolution (i.e., samples dated after this range). If a significant amount of samples is placed outside of this time frame, the timestamp might be deemed relatively inaccurate. This approach is naive and has notable limitations as it relies on the precision of the malware family labels and also the relative accuracy of the news feed. Nevertheless, it provides a good notion of the accuracy of the timestamps, especially when the results are compared.

4.2.3. Concept drift handling

The concept drift problem is usually identified with large data streams, as the meaningful properties and descriptors of data are prone to change over time (Aggarwal, 2015; Margara and Rabl, 2018). Android malware detection, which can be considered a constant flow of data, faces concept drift issues. The impact of concept drift on detection classifiers is a substantial decrease in performance over time until the detection model becomes obsolete. Thus, concept drift issues should be taken into account to design high-performance and long-lasting detection systems that are resilient to data changes over time.

In this study, an algorithm designed to handle concept drift in data streams was used to explore the impact of the timestamps to model and handle emerging data drift effectively. In Zyblewski et al. (2021), the incoming data stream is split into *chunks* and processed sequentially. The method uses a pool of classifiers trained on past data chunks to predict new data samples. During the process, the best ensemble of classifiers is dynamically selected to perform accurate predictions. Furthermore, the pool is updated after each new incoming data chunk by introducing classifiers trained on the new data and removing low-performance, *aging* models. This last step makes the system resilient to concept drift, keeping high performance over time by updating the pool of classifiers with new, evolved data.

The original algorithm by Zyblewski et al. (2021) was slightly modified for our scenario as described in the proposed methodology by Guerra-Manzanares et al. (2022a). More specifically, the initial data chunk was split to generate a full pool from the beginning. This fact avoids the need of waiting to n chunks to gradually fill the classifier pool (i.e., the n hyper-parameter refers to the fixed pool size), as the original solution proposes. This enhanced classification methodology to deal with Android data concept drift, as detailed in Guerra-Manzanares et al. (2022a), was used to analyze the impact of different timestamps on concept drift handling in distinct feature spaces within the same problem domain.

The performance of the classification method was evaluated using the $F1$ score metric. The $F1$ performance metric provides a notion of the accuracy of the classifier in detecting malware instances. It is a comprehensive metric that is calculated as the harmonic mean of *precision* and *recall* performance metrics as defined by the following equation:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{1}$$

In this regard, the *precision* of a classification model informs about the fraction of actual malware data points that the model correctly classified as *malware* among all malware predictions and is calculated as:

$$Precision = \frac{TP}{TP + FP}, \tag{2}$$

where TP refers to *True Positive*, that is, the number of actual malware that was predicted as malware by the model, and FP relates to *False Positive* or the number of benign instances incorrectly predicted as malware by the model.

The *recall* metric reports the fraction of samples classified as malware (i.e., positive) among the total number of actual malware samples included in the testing set. It is calculated as:

$$Recall = \frac{TP}{TP + FN}, \tag{3}$$

where TP relates to *True Positive* and FN refers to *False Negative*, that is, the number of actual malware instances incorrectly predicted as benign by the classification model.

In the case of imbalanced data scenarios, such as the one explored in this study, the F1 score reports better the positive class detection performance than overall accuracy, thus being a preferred performance metric for our analysis.

5. Results

5.1. Data collection

The original data set was composed of 41,382 malware samples and 36,755 benign apps. Each sample was described by 288 system call features (i.e., counts), 166 permission-related features (i.e., binary indicators), 4 timestamps (i.e., *earliest modification*, *last modification*, *first seen,* and *first seen in the wild*), the hash value, and the class and family labels. To extend the set of feature spaces evaluated, API calls were collected. The API calls features extraction was successful for 99.82% of the original data set, with the final data set composition described in Table 3.

Table 3
Final data set composition.

Class	Original data set size	Error	Final data set size
Benign	36,755	38	36,717
Malware	41,382	104	41,278
Total	78,137	142	77,995

As can be observed in Table 3, the API call collection process was not successful for 104 malware apps and 38 benign apps from the original data set due to corrupted file headers in particular *apk* archives (i.e., *zip* files), which prevented the extraction tool from parsing the compressed bundle and retrieving the data (i.e., failed *magic number* verification provoked a critical *Bad Zip File* error). As the comparative analysis required the same data set to be used in all the experimental setups, the final data set used was composed of 36,717 benign samples and 41,278 malware samples. There were no issues retrieving the *dex date* and *manifest date* timestamps for any of the samples.

Therefore, the final data set samples are described by 3 distinct multi-dimensional feature spaces (i.e., *system calls, permissions,* and *API calls*) and 6 timestamps (i.e., *last modification, earliest modification, first seen, first seen in the wild, dex date,* and *manifest date*).

5.2. Timestamps analysis and relations

The following sections describe and provide a comparative analysis of the timestamps from different perspectives regarding *availability* and *reliability* measures. This is the essential part of this study that aims to compare and evaluate distinct timestamp approaches and their suitability to build concept drift resilient machine learning-based Android malware detection models.

It is worth noticing that of the set of six timestamps analyzed in this research, two correspond to *external* timestamps, not extracted from the apk archive metadata, which were set in this case by VirusTotal scanning reports. These timestamps are the *first seen* and *first seen in the wild*. An external timestamp is less prone to be manipulated by perpetrators as it is not in the immediate scope of the attacker. However, they can be prone to delays as they depend on users' proactive behavior (i.e., user submission to VirusTotal's service) and processing errors. Besides, the *first seen in the wild* timestamp, defined as the first time the app was seen anywhere on the internet, might not be set for benign applications.

The remaining four timestamps are *internal* timestamps, collected from the inner files of the app bundle. Therefore, they can be manipulated or removed by a motivated attacker.

5.2.1. Prevalence of timestamp data

The prevalence of timestamps data provides a notion of data availability, which is critical to building effective learning systems. If the timestamp cannot be retrieved, the sample cannot be located in the historical context and is, consequently, *unusable*. Based on this, Fig. 1 conveys graphically the *prevalence* or *availability* property for every timestamp and the whole data set. The horizontal axis provides the timestamps, referenced in abbreviated form. *EM* refers to the earliest modification timestamp, *LM* to the last modification timestamp, *FS* is used for *first seen, FSW* for *first seen in the wild, DD* for *dex date,* and *MD* for *manifest date*. For every timestamp, two vertical bars are defined, which inform about the relative frequency or *percentage* of data samples that had the timestamp *available*, meaning that it was defined or properly set. The colored areas refer to class-wise proportions (i.e., red for malware,

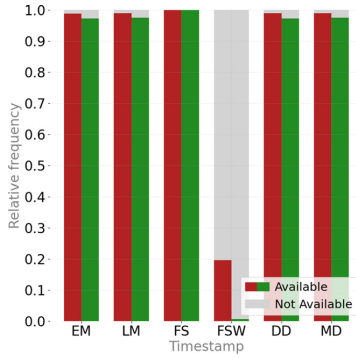


Fig. 1. *Availability* of timestamps.

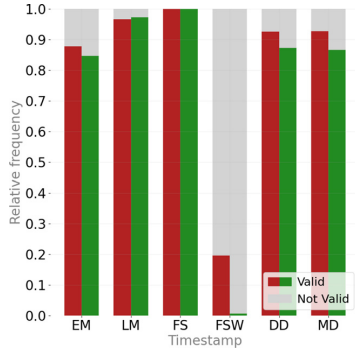


Fig. 2. *Validity* of timestamps.

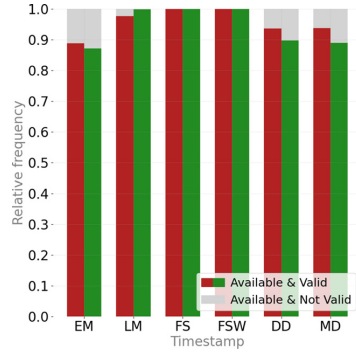


Fig. 3. *Suitability* of timestamps.

green for benign apps), while the grey areas indicate the proportion of data samples that did not have the specific timestamp available. Two main causes of *nonavailability* were found. In the case of *FSW,* the data might not be found in the detection report, so there was no timestamp set for the sample. For the other timestamps, a *nonavailable* timestamp was found as null-valued timestamp metadata, which was incorrectly parsed and retrieved by the software as *1980-00-00*.

As can be observed in Fig. 1, most of the timestamps are available for all data samples, as most of the bars reach beyond 97% prevalence. Furthermore, the *first seen* was defined for all data samples. This was expected as the timestamp is automatically set upon submission of the sample for the first time. The other timestamps, collected using different means and from the internal files, are mostly available, especially for malware instances. More precisely, a larger proportion of null-valued timestamps was found on legitimate apps. A fact that may seem counter-intuitive, but that was also pointed out by du Luxembourg (2021). An exception to the high availability of the timestamps is the *first seen in the wild* timestamp. As most of the reports retrieved did not provide data for this feature, it is missing for most of the applications, especially for benign apps (i.e., found only for 0.6% of them). This is logical as the scanning service aims to detect malware threats effectively (i.e., positive detection), so the *first seen in the wild* location for benign instances is actually irrelevant.

5.2.2. Validity of timestamps

The timestamp for any Android app sample should be located within the Android historical timeline, which encompasses from the 22nd of October 2008 (i.e., Android Google Market public release) to the present day. Any timestamp located within this time frame is deemed *valid*. Timestamps located in the future (e.g., 2107) or in the past (e.g., 1997), which were found in the data, are *impossible* configurations, suggesting tampering and consequently labeled as *not valid*. Fig. 2 reports the *validity* property for every timestamp and the whole data set. Similar to Fig. 1, the horizontal axis provides the timestamps, referenced in abbreviated form. The vertical bars report the proportion of *valid* timestamps for each class using green and red colors and the *not valid* as shaded areas.

Fig. 2 reports similar values to the ones in Fig. 1 for the *FS* and *FSW* timestamps. However, for the *EM, LM, DD,* and *MD* timestamps, the bars reach lower figures, especially for the *EM* timestamp. This indicates that this timestamp is the one that contains more non-valid values, followed by *DD* and *MD* timestamps. In all cases, except for *EM*, malware samples reach higher values than benign samples, which again might seem counter-intuitive. However, this fact may only reflect a general disregard regarding timestamps by benign apps' developers but does not provide any hint about the accuracy of the timestamp.

5.2.3. Suitability of timestamps

For the purpose of this research, the concept of *suitability* provides a notion about the most *usable* timestamps, that is, they are *available* and *valid*. Fig. 3 reports the *suitability* proportion per timestamp and class. In this case, the colored areas refer to class-wise timestamps that are both available and valid. The grey areas report the proportion of samples that have available but invalid timestamps.

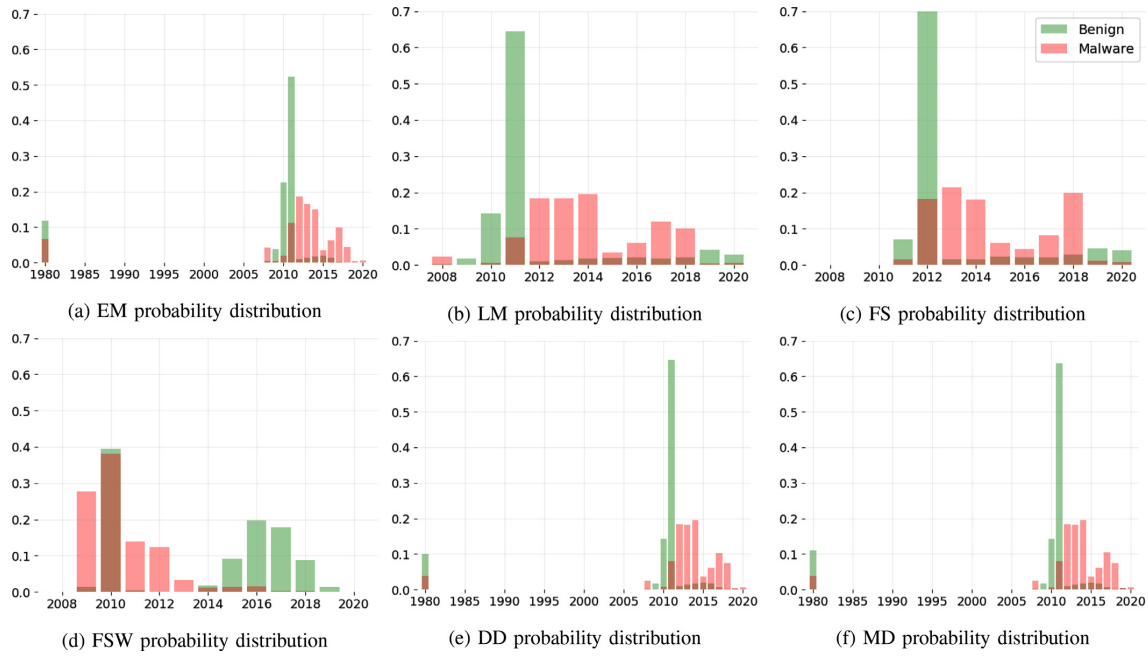
Fig. 3 conveys that *FS, FSW,* and *LM* are the most *suitable* timestamps, with a large proportion (i.e., 100% for *FS* and *FSW*) of available data that lie within the valid time frame. However, despite the high *suitability* of *FSW*, its low prevalence makes it a worse option than *FS* and *LM* if data quantity is a requirement. The *EM, DD,* and *MD* timestamps show values ranging from 87% to 93%, thus deemed as the *least suitable* options.

Figs. 1–3 enable us to rank the timestamps based on their combined properties. As a result, *FS* and *LM* significantly outperformed the other timestamps based on the analyzed criteria.

5.2.4. Variability: assessing the similarity of timestamps

The probability distribution of each timestamp (i.e., relative frequency) is provided in Fig. 4. The probability distribution is shown in a discrete representation to emphasize the differences among years. The relative frequencies are preferred to absolute values (i.e., counts in histograms) as they enable the visual comparison of the distributions. For each graph, the color of the bars refers to the class distribution (i.e., green for benign software and red for malware). The X-axis provides the year of the bar data, while the Y-axis provides the relative frequency for each year. It is worth noticing that the horizontal range is not the same for all graphs, whereas the vertical range is the same. This keeps the proportions on the Y-axis comparable while showing the whole range of values the distribution encompasses on the X-axis. The only exception is the *LM* graph, where only the range between 2008–2020 is provided due to the negligible proportion of *outliers* or non-valid values, which were not observable when plotting the whole 1980–2020 range. Furthermore, an enhanced visual comparison between *LM* and *FS* (i.e., the most *suitable* timestamps) is enabled when only the *valid* range is plotted.

As can be observed in the graphs in Fig. 4, the *internal* timestamps (i.e., *EM, LM, DD, MD)* seem to provide similar data distributions. The *LM*, however, does not show the large proportion of data points (i.e., around 10%) located in 1980 that the other *external* distributions have, but the distributions in the valid range



(a) EM probability distribution (b) LM probability distribution (c) FS probability distribution

(d) FSW probability distribution (e) DD probability distribution (f) MD probability distribution

Fig. 4. Probability distribution of each timestamp.

are relatively similar, especially when compared with *DD* and *MD*. It is worth noticing that due to different X-axis ranges the visual comparison is hindered. The *FSW* data distribution is radically different from the other distributions, showing malware data concentrated in the 2008–2016 range and legitimate data in the 2014–2019 range and also in 2010. An interesting observation occurs when *LM* and *FS* are compared, which motivates the plotting of *LM* only for the *valid* range. As can be observed, the two distributions seem relatively similar for legitimate data, peaking in one year and showing relatively low figures for the other years. However, the peak occurs in 2011 for *LM* and 2012 for *FS*. In the case of malware, the three consecutive bars around 0.2 probability value occur in both distributions in the range 2012–2014. However, the dispersion of data surrounding this range is distinct, with many samples in the years before this range for *LM* and in the years after this range, especially in 2018, for *FS*. These observations may suggest that the relatively similar but shifted shapes might have been caused by a *delay* in the *FS* timestamp regarding the *LM* timestamp. Further exploration of this hypothesis is addressed in the following paragraphs, using statistical means, and in Section 5.2.6.

The statistical analysis of timestamps distributions enables the assessment of their similarity, which provides a notion of the degree of variability among them. For this purpose, *Jensen–Shannon distance* (i.e., JSD) and *Kolmogorov–Smirnov 2-sample test* (i.e., KS) were used. The former uses the notion of distance between distributions to provide a similarity score, bounded in the [0, 1] interval, where higher values report greater dissimilarity, while the latter compares the experimental cumulative density distributions to statistically infer if the distributions are likely to belong to the same population, thus being similarly distributed. In this case, the *p-value* is used to assess the statistical significance of the results by accepting or rejecting the null hypothesis (i.e., the distributions are equal) at a specific confidence level α. Thus assessing the similarity between the distributions. In general, small *p-values* indicate a high probability that the distributions come from the same population, thus reporting a greater similarity between the timestamps to locate the data within the Android historical timeline.

As a result, both metrics provide complementary measurements to assess the similarity of the distributions. If distributions are similar, we would expect a *JSD* value close to 0 and a *KS* value close to 1. If they are significantly different, the *JSD* value is expected to be close to 1 and the *KS* value close to 0. Both measures are symmetric, meaning that the order used to compare the distributions does not matter (i.e., $d(P, Q) = d(Q, P)$).

The matrix in Fig. 5 provides the comparison among all pairs of timestamp distributions for the benign data. Given the symmetry of the calculated measures, they enable us to provide the computed values for both similarity measures in the form of a matrix where the main diagonal is left blank to separate the values. The values above the diagonal of the matrix provide the values for *KS* computations, while the values below the diagonal provide the obtained *JSD* values.

As can be observed in Fig. 5, all the timestamps seem to provide different distributions for the data. The only exception is for the pair *DD-MD*, which has an *almost* 0 distance (i.e., *almost* perfect similarity) and a KS of 1. These values strongly suggest that these two distributions are roughly the same. This fact was also spotted on the graphical visualization in Fig. 4. Furthermore, *DD* and *MD* have a high degree of similarity (i.e., they show a small distance and large KS value) with *LM* and, to a lesser extent, with *EM*. This fact shows the relatively close distance between the distributions based on *internal* timestamps, which confirms the spotted similarities in the plots. However, *EM* appears to have a significantly different cumulative function as reflected by the small *p-values*, probably caused by the large number of *invalid* values included in this distribution and the higher peaks in the early years (i.e., before 2015) observed in the graphical depiction of the distribution. In the case of *DD* and *MD*, their distributions can be interpreted as *almost* equivalent. Nevertheless, *LM* is preferred as it provides

Fig. 5. JSD-KS matrix for benign data.

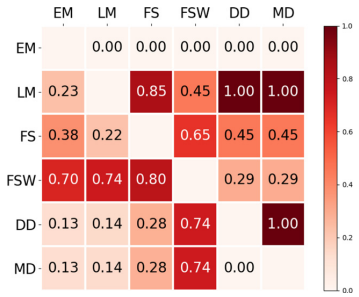


Fig. 6. JSD-KS matrix for malware data.

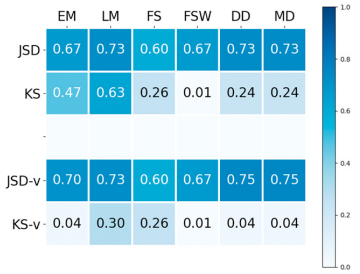


Fig. 7. JSD-KS inter-class.

more *suitable* timestamps than *DD* and *MD*. Regarding the *external* timestamps, *FS* and *FSW* show unique distributions, as evidenced by the large distance values and small p-values with almost all the other distributions. Despite having the same origin (i.e., VirusTotal), they are significantly different from each other and all other distributions, especially in the case of *FSW* (i.e., all *p-values* are near 0). When the most *suitable* timestamps are compared, *FS* and *LM* seem to have a great dissimilarity (i.e., a distance of 0.77, so they locate samples differently) but show a relatively low p-value (i.e., 0.21), which confirms the interesting observation from the graphs. Their cumulative functions are close enough to be found relatively similar using the KS-test but remarkably different when the distance-based similarity is used.

The matrix in Fig. 6 provides the comparison among all pairs of timestamp distributions for the malware data. The area above the diagonal provides the KS values, while the area below the diagonal provides the JSD values.

As can be seen in Fig. 6, the overall situation is similar to the benign case. The *external* timestamps show the greatest similarity, with *DD* and *MD* being almost identical to each other and also to *LM*. Again, *EM* shows to be relatively different from all other distributions, emphasized by the large proportion of data points in the *invalid* range. The *external* timestamps differ significantly, but less than for the benign data. An important exception to the similar trends with the benign data emerges when comparing *FS* and *LM*. For the malware case, these two distributions have a smaller distance and a significantly higher *p-value*. This fact shows that the range and overall shape of the distribution are similar and also that the cumulative function of the distributions is relatively similar. This fact supports the hypothesis of the *delay* between them and that even though they may provide similar distribution of malware data along the historical timeline, there is more concentration of samples in the early years for *LM* and in the latter years of the valid range for *FS*.

The previous statistical analysis compared the distribution of data according to timestamps for the same class (i.e., benign and malware). An interesting comparison is also the analysis of the similarity of class distributions within a given timestamp. The results are reported in Fig. 7, where the columns provide information about the specific timestamp and the horizontal rows about the statistical value computed when comparing the benign and malware distribution for that specific timestamp (i.e., *JSD* or *KS*). The upper rows in the figure provide the comparative results of *JSD* and *KS* for the whole time frame (i.e., whole distributions), whereas the lower rows provide the same information but just for the *valid* time frame, indicated as *JSD-v* and *KS-v*.

The overall interpretation of Fig. 7 is that the *valid* time frame emphasizes the differences between class distributions. In general, the values of distance increase and *p-values* diminish in the lower rows (i.e., *valid* range) compared with the upper rows (i.e., including the *invalid* range). The only exceptions to this are the *FS* and *FSW* timestamps which have the same values on both pairs of rows as they are always *valid*. Therefore, the class-wise distributions are significantly different across all timestamps.

5.2.5. Accuracy: a measure of historical context reliability

The evaluation of timestamp accuracy (i.e., how precise the location of a sample is within the Android historical timeline) is a significant challenge due to the absence of an exact ground-truth timestamp (i.e., it is not possible to ascertain when the malware sample was first released). In this regard, only approximations to the ground-truth timestamp can be achieved. This approximation might be based on external information such as open-source intelligence (OSINT) feeds, discovery reports of specific malware families by antivirus vendors, or media news. Therefore, the assessment of timestamps' accuracy and reliability is hindered and can only be approximated using these OSINT sources, which might be relatively *delayed* and not fully precise. In this research, this approach was used to obtain an approximation of the reliability of the timestamps evaluated. In this regard, the first *discovery* report released for specific malware families was used to contextualize each malware family within the historical timeline. The first two columns in Table 4 show 10 malware families (i.e., one per row) and *reference* dates as a context of the discovery time frame based on the reports (i.e., month/year). The data sources are provided in square brackets. They were taken from reliable sources when possible and contrasted with other media feeds. The following 6 columns are split into three sub-columns which are referred to as ± 6, > 6, and NV. For the sake of interpretation of the table, these columns have been colored in green, yellow, and grey, respectively. These columns show the proportion of data samples (i.e., percentages) of the data set dated with each specific timestamp that *lies within the reference value* ± 6 *months* (i.e., ± 6 column), *beyond the reference value + 6 months* (i.e., >6 column) and that has a *non-valid* location (i.e., *NV* column). Note that the values *before the reference time - 6 months* are not provided but can be computed by summing the provided proportions and subtracting to 100. The rationale behind these computed proportions is the following. A precise timestamp should locate most of the samples of a specific family in the ± 6 and > 6 range, which is defined as the *valid* range for the specific family. The proportion of NV values and samples located *before* the valid range should be minimal (i.e., ideally zero). A larger proportion of values in the > 6 range may imply a *delay* in the timestamp or denote family evolution. The ± 6 range gives a notion of the amount of samples within this range, but due to family evolution it can only be interpreted in comparison with the other values, as the data set may contain fewer *original* samples than evolved samples. Furthermore, malware family denominations are not consistent among AV vendors or even an-

Table 4
Accuracy analysis of timestamps to locate specific malware families.

Family	Reference	EM			LM			FS			FSW			DD			MD		
		6	> 6	NV	6	> 6	NV	6	> 6	NV	6	> 6	NV	6	> 6	NV	6	> 6	NV
Geinimi	11/10 [90]	31.3	43.8	6.3	56.3	43.7	0	12.5	87.5	0	18.8	0	56.3	56.3	43.7	0	56.3	43.7	0
DroidDream	03/11 [91]	65.9	19.8	0	67	28.6	0	14.3	85.7	0	16.5	4.4	61.5	67	28.6	0	67	28.6	0
DroidKungFu	06/11 [92]	66.2	13.1	11.9	68	31.6	0.3	7.6	92.4	0	2.5	16.5	54	68.3	31.4	0.3	68.3	31.4	0.3
Plankton	06/11 [93]	23.8	71.4	1.6	22.2	77.8	0	6.3	93.7	0	0	1.6	92.1	22.2	77.8	0	22.2	77.8	0
GinMaster	08/11 [94]	21.2	70.4	6.6	17.6	81.5	0.2	0.7	98.9	0	5.2	3.9	69.8	18.4	80.7	0.2	18.4	80.7	0.2
AnserverBot	09/11 [92]	96.3	0	1.0	99.7	0	0	42.1	57.9	0	0.3	47.2	39.1	99.7	0	0	99.7	0	0
Slocker	05/14 [95]	23.2	50.3	25.8	23.1	57.9	18.4	1.1	98.5	0	0.1	1.1	93.9	23.2	55.1	21.1	23.3	54.4	21.7
MobiDash	01/15 [96]	3.3	60.1	36.7	3.3	90.2	6.5	0.7	99.3	0	0.7	1.3	96.7	3.3	60.8	35.9	3.3	60.1	36.6
BankBot	01/16 [97]	60.4	17.3	12.3	60.5	20.4	9.2	71.3	27	0	4	0.2	76	60.5	20.2	9.4	60.5	19.6	10
Triada	03/16 [98]	41.7	0	58.3	41.7	16.7	41.6	41.7	58.3	0	41.7	8.3	50	41.7	16.7	41.6	41.7	16.7	41.6
Total	-	43.3	34.6	16.1	45.9	44.9	7.6	19.8	80	0	9	8.5	69	46	41.5	10.9	46	41.3	11

References [90–98] citation mentioned in Table 4 (F-secure, 2021b; F-secure, 2021a; Jiang and Zhou, 2013; Shipman, 2011; Yu, 2013; Jiang and Zhou, 2013; Lipovsky et al., 2017; Kiss et al., 2016; Dr.Web, 2018; Buchka and Kuzin, 2016).

alysts, thus being a handicap for any malware family analysis. In our case, it is assumed that most of the labels are *certain*, which provides a relative degree of flexibility in interpreting the results. The last row of the table computes the *totals* or average value of each column for each specific timestamp.

As can be observed in Table 4, the individual proportions for specific malware families greatly vary among timestamps. For instance, *AnserverBot* seems to be well captured by all external timestamps in the reference ±6 months time frame, which may imply that the *reference* might be precise for the family discovery date and corresponding outbreak, and that these timestamps precisely locate this malware family. A different situation happens with the *MobiDash* family, where almost all timestamps convey the idea that the outbreak of this family happened in a later time frame, but also that the initial reference might be precise as the sum of the three proportions is 100% (i.e., no samples *dated* before the reference time frame). However, in general, individual values greatly vary across families and timestamps. A better and broader picture is provided by the *total* values. Even though the average value might be significantly biased due to *outlier* values, it is a good indicator of the overall trend. Based on the interpretation of the *total* values, it can be stated that the *LM* timestamp provides the most desirable properties of accuracy. It has a very low ratio of *non-valid* values and a high proportion of timestamps within the *valid* time frame (i.e., sum of values in the time frame encompassed by *reference* ±6 and > 6). The average values also confirm that the internal timestamps show similar distributions, with all of them showing similar proportions but with significantly lower *non-valid* values for the *LM* timestamp. The external timestamps show completely different pictures. The *FS* is characterized by always providing *valid* values, whereas the *FSW* shows a large proportion of *non-valid*, which correspond to *missing* data in this case and not actual non-valid values. Finally, when the *FS* and *LM* timestamps are compared, the average values show that *FS* captures most of the data beyond the *reference + 6 months* (i.e., > 6), whereas *LM* does it in similar proportions on both *valid* time frames. This supports the delayed nature of *FS* to capture malware outbreaks and the goodness of *LM* to locate most data samples with improved precision.

The results shown in Table 4 and the analysis performed suggest that *FS* and *LM* are significantly better timestamps than the other analyzed approaches in terms of suitability, statistical properties, and accuracy. To further investigate their relationship, the next section analyzes statistically the differences between them and their relation over time.

5.2.6. Last modification vs. first seen: a comparative analysis

Figs. 8 and 9 provide the differences between both timestamps, computed for each data sample, separately for benign and malware data. The base *difference* unit is *days* and the base timestamp used is the *last modification*, so that the differences can be expressed in positive terms (e.g., +8 days). The assumption was that the *last modification* timestamp would place the sample more accurately in the Android historical timeline, usually earlier than *first seen*. Therefore, it was chosen as the reference time. These graphs report relevant descriptive statistics regarding the temporal differences (i.e., *Y*-axis) for the samples in the data set located in a specific period (i.e., six months chunks) using the *last modification* timestamp (i.e., *X*-axis) concerning the *first seen* timestamp. The data is split into chunks of 6 months data (i.e., period) for better interpretation of the results and deeper exploration of the differences. Only the *valid* time frame is plotted (i.e., from 2009 to

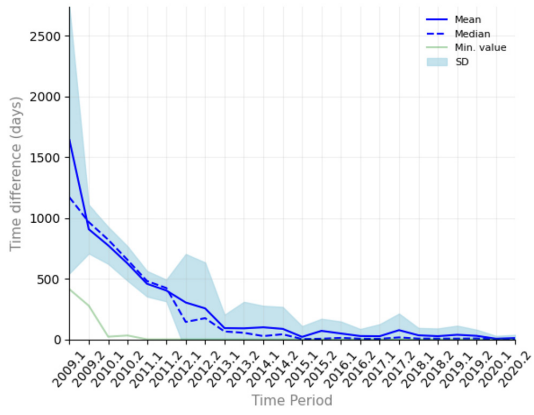


Fig. 8. Differences *LM-FS* for benign data.

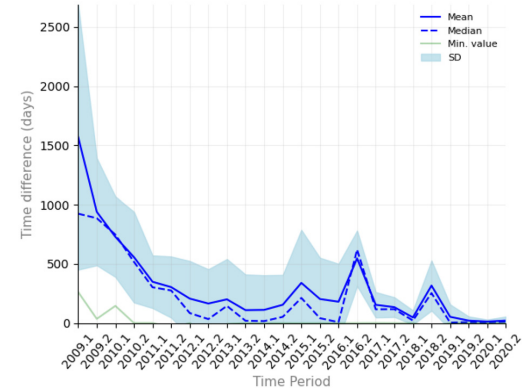


Fig. 9. Differences *LM-FS* for malware data.

2020). The semesters are referenced as appended suffixes to the year figure (e.g., *2009.1* reflects the first six months of 2009). The temporal differences are calculated individually per data sample and grouped into data periods. Descriptive statistics are calculated per period: *mean*, *median*, *minimum difference*, and *standard deviation*. In this regard, the solid blue line reports the *average* value for each period, while the dashed blue line provides the *median* value. These two central tendency measures report the average value of displacement of a sample per period. The green line provides the *minimum difference* value found in each period. The standard deviation, plotted as a blue area, conveys the average spread of differences around the mean for the data samples located in each chunk.

As expected, in both cases, a positive difference between both timestamps is observed. This evidences that the *first seen* timestamp locates the samples later in the timeline, thus *delayed* regarding the *last modification* timestamp. This fact is especially pronounced in the early years of Android history, with average differences of around 1500 days (i.e., four years) for both malware and benign applications. Thus it implies that an instance located in 2009.1 (i.e., the first semester of 2009) by the last modification timestamp might be located by the first seen timestamp in 2013.1 (i.e., the first semester of 2013). This significant difference in the temporal location of samples might impact the performance and adaptation capabilities of a detection system to deal with concept drift, as *first seen* may generate *artificial drift* by misplacing the data, which might be more complex to model effectively than real concept drift, generally smoother.

However, as can be observed in Figs. 8 and 9, the differences between these timestamps have been reducing over time, as evidenced by the monotonically decreasing mean value for benign instances and the significant decrease that has occurred in the case of malware instances, especially in recent years. This fact has made the timestamps more synchronized and closer over time, and even equivalent for the 2019–2020 time frame. For instance, 2020.1 and 2020.2 periods have mean values of 4.88 and 12.37 days and median values of 2 and 3 days, respectively, for benign samples, and average values of 15.9 and 16.45 days and medians of 5 and 11 days, respectively, in the case of malware samples. This is a dramatic change when compared to 2010.1, which shows a mean value of 728.4 days and a median of 747 days for malware, and an average of 774.3 days and a median of 821 days for benign data. As a result, the gap between both timestamping approaches to date samples has significantly decreased over time, making them converge and, consequently, increasing the reliability and accuracy of the *first seen* timestamp in more recent years (i.e., 2019–2020).

The convergence of both timestamps supports the hypothesis that the *last modification* timestamp is accurate and that it is *rarely* tampered with by attackers. Consequently, if the system has to learn from past data and predict about past samples, it might be safer to use the *last modification* timestamp, whereas, if the system uses mainly recent data, the convergence of the timestamps implies that both approaches could be appropriate and perform similarly against data drift. Furthermore, if data tampering is a major concern, the usage of *first seen* ensures that the data have not been tampered with, even though a delay should always be assumed.

5.3. Concept drift handling

For the purpose of this research, the existence of concept drift is assumed and not proven. The concept drift phenomenon has already been proved and explored in previous research for the feature spaces used in this study: system calls, permissions, and API calls (Guerra-Manzanares et al., 2022a; Onwuzurike et al., 2019; Xu et al., 2019).

Table 5
Data set size per timestamp in the period 2011.2-2018.1.

Timestamp	Malware	Benign	Total
EM	33,346	7602	40,948
LM	38,496	13,456	51,952
FS	40,376	32,870	73,246
FSW	2,137	116	2,253
DD	36,805	11,555	48,360
MD	36,810	11,500	48,310

The purpose of this experimental scenario was to evaluate the impact and adaptive response generated by the appearance of concept drift, which is distinct for different timestamps and feature spaces, and analyze what approach was better to model *natural* concept drift, which is assumed to be mostly a relatively *smooth* transition with eventual *sudden* drifts. The sudden changes may occur, for instance, when a new malware outbreak occurs and the ML model has to learn about the new threat, which is not similar to previous data, and, consequently, adapt to it by updating its knowledge.

The *adaptive* classifier used in this study requires the selection of hyper-parameters such as the *chunk size* (i.e., number of samples per chunk), *pool size* (i.e., number of classifiers in the pool), and *time constraint* (i.e., max. time frame of data included in the chunks). Based on the data distribution and experimental tests, a good set of hyper-parameters was 4000 samples per chunk, 12 classifiers in the pool, and 3 months of data per chunk. All the classifiers in the pool were Random Forest models, which have proved successful for the task in similar studies (Guerra-Manzanares et al., 2019b). As the data set is imbalanced towards the malware label, the random oversampling technique was applied to the training chunks to balance the classes and avoid a biased classifier.

Due to the characteristics of *KronoDroid* data and the demands of machine learning models, it was not possible to use the whole Android historical timeline to build effective ML-based classifiers per quarter (i.e., there is not enough data in the early years or more recent years), so the experimental setup was restricted to the period encompassing from the second semester of 2011 until the first semester of 2018. This time frame spans 7 years of Android history, including the most active years of Android malware development (AV-Test, 2021; Johnson, 2021).

The available data per timestamp for the selected period (i.e., from 2011.2 to 2018.1) is provided in Table 5.

As can be noticed in Table 5, *FS* provides most of its data within this range, whereas the external timestamps provide lower proportions of samples (i.e., especially in the case of *EM*). As expected, the data is imbalanced towards the malware class, thus justifying the usage of a data set balancing technique to avoid any class bias from the classifier. Finally, as the data provided by the *FSW* timestamp is not enough to build a single classifier, this timestamp was discarded and not used in the following experimental setups.

Three experimental setups are described in the following sections. The distinct feature spaces were explored individually and used as input features to the *adaptive* classifier algorithm. The data was split into chunks and processed sequentially. The timestamps were used to place the data samples into consecutive chunks, and the system performance was retrieved. The *F1* score performance metric is provided for each induced *adaptive* classification model (i.e., one per each combination of a timestamp and feature space). It is worth noticing that no hyper-parameter optimization was performed as the main aim was to test the same scenario for all timestamps and compare the outcome, so no model was optimized to ensure the same conditions. The performance graphs provide each model performance with different color and/or line style. *EM*,

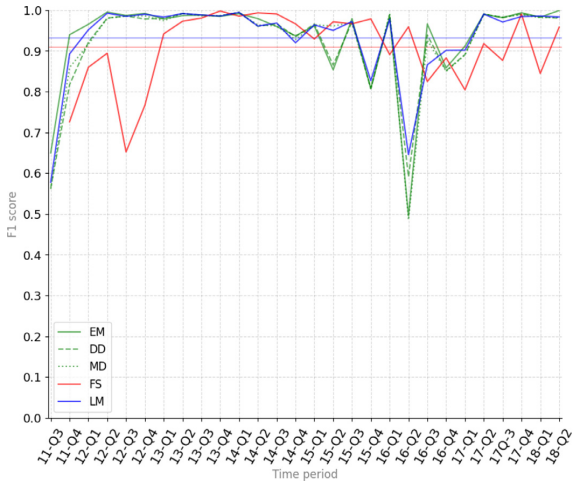


Fig. 10. F1 performance of permissions-based models using different timestamps.

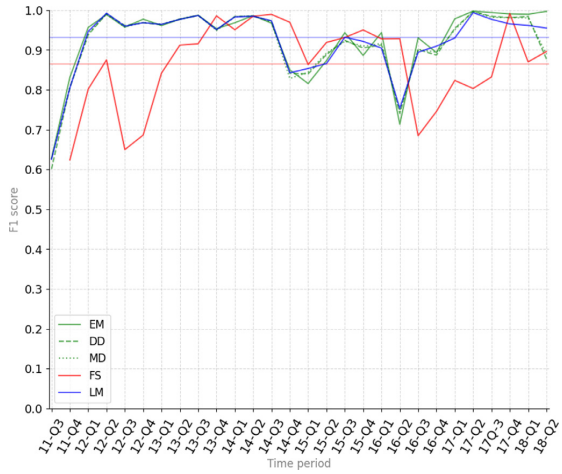


Fig. 11. F1 performance of system calls-based models using different timestamps.

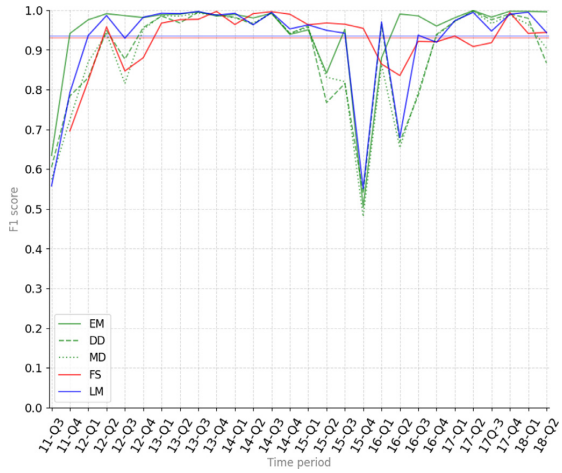


Fig. 12. F1 performance of API calls-based models using different timestamps.

DD and *MD* are provided in green color but with solid, dashed, and dotted line styles, respectively. *LM* is reported with a solid blue line while *FS* with a solid red line. As the most *suitable* options were *LM* and *FS*, their average performance for the whole period is reported with a horizontal overlay line, *LM* with a solid blue line and *FS* with a solid red line.

5.3.1. Permissions

The performance results of the models built for the permissions feature space and using the different timestamps are provided in Fig. 10. The permissions feature space is defined by categorical data and is the smallest of the analyzed ones (i.e., 166 dimensions). As can be observed, despite two sudden drops (and consequent recovery), the smoother line is provided by the *LM* timestamp. The other external timestamps provide similar performance but describe a more fluctuating surface. The *FS* timestamp performance is not smooth, characterized by several sudden peaks and bottoms in neighboring quarters, especially at the beginning and the end of the analyzed time frame (i.e., bumpy red line). Furthermore, it has the lowest average in the analyzed period. Despite that, the overall performance of all models is over 0.90 F1, which reflects the goodness of the system to deal with concept drift, and especially with *natural* data drift, which is better captured by the *internal* timestamps.

5.3.2. System calls

The performance of the models built using the system calls feature space is provided in Fig. 11. The system calls feature space is numeric and larger than the permissions space (i.e., 288 dimensions). Similar to Fig. 10, the external timestamps provide smoother performance lines, and, again, the *LM* timestamp seems to yield the best performance. *LM* enables the model to capture better the natural data drift, showing quick recovery after sudden data drifts. However, in this case, *EM* achieves similar performance over the whole range but shows a more irregular performance line. As in the permissions space, *FS* provides the worst performance and is characterized by sudden dips and peaks, likely caused by *artificial* data drift. An interesting fact in this plot is that from 13-Q3 until 16-Q3, *LM* and *FS* seem to perform synchronously. The *FS* performance line is relatively similar but delayed one quarter regarding *LM* and reaches more extreme values. This goes in line with the *median* differences observed in this time frame in Figs. 8 and 9 (i.e., below 90 days for both classes). Lastly, in this case, the difference between *FS* and *LM* average performance is significantly larger, with the average performance of *LM* of around 0.93 and at the 0.87 level for *FS*.

5.3.3. API calls

The performance of the models built using the API calls feature space is provided in Fig. 12. The API calls feature space is the largest feature space, with over 53,523 features. Similar to the other feature spaces, the performance of the internal timestamps is rather similar, and over the performance of *FS*. However, in this case, two deep dips are observed for the internal timestamps that are not observed for *FS*. It is worth noticing that, in the case of high-dimensionality spaces, the quantity of data is critical to building effective models (i.e., data density is needed to generate precise classification boundaries), a phenomenon called the *curse of dimensionality*. As reflected in Table 5, the data samples available for the internal timestamps are fewer and significantly reduced for these specific periods. Therefore, reduced performance might be observed due to insufficient data to cover the feature space effi-

ciently. However, despite the deep dips in those specific periods, the average performance of *LM* still outperforms *FS*.

6. Discussion

Timestamp selection has not been explored in concept drift-related research, where some common approaches are usually used without considering alternatives. However, depending on the timestamp selected, the same app might be located in different temporal contexts within the Android historical timeline. The position of the apps along the timeline determines the observed drift, whether natural or artificial. Therefore, timestamp selection is critical to handle concept drift effectively.

This research has demonstrated that the selection of the timestamp is an important decision in building long-lasting machine learning-based Android malware detection systems that adapt and learn over time about drifting data. The most common approaches for timestamps used in research studies, such as *FS* and *DD*, have been shown sub-optimal in tackling emerging concept drift. Furthermore, the test scenarios and statistical analysis suggest that *FS* should not be employed when *old* data is used, as it can misplace data samples along the historical timeline, generating artificial data drift that cannot be modeled efficiently by machine learning systems due to the randomness and noise added to the detection task. In the seek for alternatives, *LM* has emerged as a reliable and suitable timestamp that outperforms other options to handle concept drift in the Android malware detection task. Despite showing an inherent tampering risk, as it is extracted from the inner files of the *apk* and, consequently, in the scope of the attacker, it has been demonstrated to be robust and accurate. Other alternatives such as *DD, MD,* and *EM* can provide similar performance, but as their suitability is lower, fewer data samples can be used to address the drift, thus providing less effective results. Furthermore, the usage of specific file-related timestamps might be riskier as there is no guarantee that the timestamps are set for those specific files (du Luxembourg, 2021). A potential benefit of the usage of *LM* or even *EM* is that they do not rely on a single file to date the application but on any of the inner files, which may overcome the single-file dependency of the *DD* and *MD* timestamps.

The implications of our findings are relevant for any Android malware detection solution seeking long-term reliable performance. Most research studies focus on the optimization of their systems in a restricted testing set, disregarding temporal context and data evolution. However, the features used for Android malware detection are likely to suffer concept drift at any point, as the dynamism of the malware generation phenomenon (e.g., new malware trends or new malware capabilities) and the evolution of the Android framework (e.g., changes in API calls or permissions) are directly reflected in the features. Thus optimizing for static historical data sets does not guarantee effective future performance. The system might not be able to predict well future and *unknown* data, which is the main objective of machine learning-based detection systems. Furthermore, the traditional random split of data into testing/training sets generates impossible data configurations and yields over-inflated performance metrics caused by subtle but harmful *data snooping*.

This research has shown that, in general, inner timestamps might be more appropriate to build models in which past data is involved, whereas, for recent data, *FS* might be a good approach as the temporal gap between samples has been reducing over time, thus increasing the accuracy of *FS*. In any case, timestamp selection is critical to achieving long-lasting, high-performance Android malware detection systems.

The main limitation of our study is that the results strictly rely on the reliability of the data used. Even though the size of the data set is large enough to enable statistical inference, the *soft* labels of the data set were used, which are not as *certain* as the *hard* labels. We chose the *soft* label to perform our analysis as more data was available, so the error due to the class label might be minimized by increased data quantity. Furthermore, the malware family analysis relied on the malware family label provided by the data set (Guerra-Manzanares et al., 2021). However, malware family naming conventions vary significantly among antivirus vendors. Therefore, the accuracy of the malware family label is not fully guaranteed. To minimize this issue, for the malware family analysis, old and well-known malware families were used as it might increase the certainty around the actual malware family label for the sample.

7. Conclusions

The vast body of Android malware detection research has neglected the impact of concept drift in Android malware research. Timestamps, critical elements for concept drift handling, have not received the deserved attention in the related literature. Our study performed an extensive benchmarking about timestamp options and their capabilities to deal with concept drift effectively in different feature spaces. Our results show that timestamp selection is a critical decision and that the *last modification* and *first seen* timestamps are the best options to build effective, long-lasting ML models for Android malware detection under data evolution challenges.

Declaration of Competing Interest

We wish to confirm that there is no known conflict of interest associated with this publication and there has been no significant support for this work that could have influenced its outcome.

CRediT authorship contribution statement

Alejandro Guerra-Manzanares: Conceptualization, Methodology, Formal analysis, Software, Validation, Investigation, Visualization, Writing – original draft, Writing – review & editing. **Hayretdin Bahsi:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing.

References

Abderrahmane, A., Adnane, G., Yacine, C., Khireddine, G., 2019. Android malware detection based on system calls analysis and CNN classification. In: 2019 IEEE Wireless Communications and Networking Conference Workshop (WCNCW). IEEE, pp. 1–6.

Afonso, V.M., de Amorim, M.F., Grégio, A.R.A., Junquera, G.B., de Geus, P.L., 2015. Identifying android malware using dynamically obtained features. J. Comput. Virol. Hack. Tech. 11 (1), 9–17.

Aggarwal, C.C., 2015. Data Mining: The Textbook. Springer.

Ahsan-Ul-Haque, A., Hossain, M.S., Atiquzzaman, M., 2018. Sequencing system calls for effective malware detection in android. In: 2018 IEEE Global Communications Conference (GLOBECOM). IEEE, pp. 1–7.

Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y., 2015. Are your training datasets yet relevant? In: International Symposium on Engineering Secure Software and Systems. Springer, pp. 51–67.

Alzaylaee, M.K., Yerima, S.Y., Sezer, S., 2017. Emulator vs. real phone: android malware detection using machine learning. In: Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics, pp. 65–72.

Alzaylaee, M.K., Yerima, S.Y., Sezer, S., 2020. Dl-droid: deep learning based android malware detection using real devices. Comput. Secur. 89, 101663.

Amin, M.R., Zaman, M., Hossain, M.S., Atiquzzaman, M., 2016. Behavioral malware detection approaches for android. In: 2016 IEEE International Conference on Communications (ICC), pp. 1–6. doi:10.1109/ICC.2016.7511573.

Android, 2021a. App manifest overview. https://developer.android.com/guide/topics/manifest/manifest-intro.

Android, 2021b. Package index. https://developer.android.com/reference/packages.

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., Rieck, K., 2020. Dos and don'ts of machine learning in computer security. arXiv preprint arXiv:2010.09470

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C., 2014. Drebin: effective and explainable detection of android malware in your pocket. In: Ndss, vol. 14, pp. 23–26.

AV-Test, 2021. Malware. https://www.av-test.org/en/statistics/malware/.

Barbero, F., Pendlebury, F., Pierazzi, F., Cavallaro, L., 2020. Transcending transcend: revisiting malware classification with conformal evaluation. arXiv preprint arXiv:2010.03856

Bhatia, T., Kaushal, R., 2017. Malware detection in android based on dynamic analysis. In: 2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security), pp. 1–6. doi:10.1109/CyberSecPODS.2017.8074847.

Buchka, N., Kuzin, M., 2016. Attack on zygote: a new twist in the evolution of mobile threats. https://securelist.com/attack-on-zygote-a-new-twist-in-the-evolution-of-mobile-threats/74032/.

Buczak, A.L., Guven, E., 2015. A survey of data mining and machine learning methods for cyber security intrusion detection. IEEE Commun. Surv. Tutor. 18 (2), 1153–1176.

Burguera, I., Zurutuza, U., Nadjm-Tehrani, S., 2011. Crowdroid: behavior-based malware detection system for android. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, pp. 15–26.

Cai, H., 2020. Assessing and improving malware detection sustainability through app evolution studies. ACM Trans. Softw. Eng. Methodol. (TOSEM) 29 (2), 1–28.

Cai, H., Fu, X., Hamou-Lhadj, A., 2020. A study of run-time behavioral evolution of benign versus malicious apps in android. Inf. Softw. Technol. 122, 106291. doi:10.1016/j.infsof.2020.106291. https://www.sciencedirect.com/science/article/pii/S0950584920300410

Cai, H., Meng, N., Ryder, B., Yao, D., 2019. Droidcat: effective android malware detection and categorization via app-level profiling. IEEE Trans. Inf. Forensics Secur. 14 (6), 1455–1470. doi:10.1109/TIFS.2018.2879302.

Cai, L., Li, Y., Xiong, Z., 2021. Jowmdroid: android malware detection based on feature weighting with joint optimization of weight-mapping and classifier parameters. Comput. Secur. 100, 102086.

Cai, L., Li, Y., Xiong, Z., 2021. Jowmdroid: android malware detection based on feature weighting with joint optimization of weight-mapping and classifier parameters. Comput. Secur. 100, 102086.

Canfora, G., Medvet, E., Mercaldo, F., Visaggio, C.A., 2015. Detecting android malware using sequences of system calls. In: Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, pp. 13–20.

Casolare, R., De Dominicis, C., Iadarola, G., Martinelli, F., Mercaldo, F., Santone, A., 2021. Dynamic mobile malware detection through system call-based image representation. J. Wirel. Mob. Netw. Ubiquitous Comput. Dependable Appl. 12 (1), 44–63.

Da, C., Hongmei, Z., Xiangli, Z., 2016. Detection of android malware security on system calls. In: 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), pp. 974–978. doi:10.1109/IMCEC.2016.7867355.

Desnos, A., Gueguen, G., Bachmann, S., 2018. Androguard. https://androguard.readthedocs.io/en/latest/index.html.

Dimjašević, M., Atzeni, S., Ugrina, I., Rakamaric, Z., 2016. Evaluation of android malware detection based on system calls. In: Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics, pp. 1–8.

Dr.Web, 2018. Doctor web: banking trojan android.bankbot.149.origin has become a rampant tool of cybercriminals. https://news.drweb.com/show/?i=11772.

F-secure, 2021a. Trojan:android/droiddream.a. https://www.f-secure.com/v-descs/trojan_android_droiddream_a.shtml.

F-secure, 2021b. Trojan:android/geinimi. https://www.f-secure.com/v-descs/trojan_android_geinimi.shtml.

Feizollah, A., Anuar, N.B., Salleh, R., Wahab, A.W.A., 2015. A review on feature selection in mobile malware detection. Digit. Invest. 13, 22–37.

Feng, P., Ma, J., Sun, C., Xu, X., Ma, Y., 2018. A novel dynamic android malware detection system with ensemble learning. IEEE Access 6, 30996–31011. doi:10.1109/ACCESS.2018.2844349.

Ferrante, A., Medvet, E., Mercaldo, F., Milosevic, J., Visaggio, C.A., 2016. Spotting the malicious moment: characterizing malware behavior using dynamic features. In: 2016 11th International Conference on Availability, Reliability and Security (ARES). IEEE, pp. 372–381.

Frenklach, T., Cohen, D., Shabtai, A., Puzis, R., 2021. Android malware detection via an app similarity graph. Comput. Secur. 109, 102386.

Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., Bouchachia, A., 2014. A survey on concept drift adaptation. ACM Comput. Surv. (CSUR) 46 (4), 1–37.

Gao, H., Cheng, S., Zhang, W., 2021. Gdroid: android malware detection and classification with graph convolutional network. Comput. Secur. 106, 102264.

Google, 2008. Android market: now available for users. https://android-developers.googleblog.com/2008/10/android-market-now-available-for-users.html.

Google, 2021. Google play protect. https://developers.google.com/android/play-protect.

Guerra-Manzanares, A., Bahsi, H., Nõmm, S., 2021. Kronodroid: time-based hybrid-featured dataset for effective android malware detection and characterization. Comput. Secur. 102399.

Guerra-Manzanares, A., Bahsi, H., Nõmm, S., 2019a. Differences in android behavior between real device and emulator: a malware detection perspective. In: 2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS), pp. 399–404. doi:10.1109/IOTSMS48152.2019.8939268.

Guerra-Manzanares, A., Luckner, M., Bahsi, H., 2022a. Android malware concept drift using system calls: detection, characterization and challenges. Expert Syst. Appl. 117200. doi:10.1016/j.eswa.2022.117200. https://www.sciencedirect.com/science/article/pii/S0957417422005863

Guerra-Manzanares, A., Luckner, M., Bahsi, H., 2022b. Concept drift and cross-platform behavior: challenges and implications for effective android malware detection. Comput. Secur. 120, 102757.

Guerra-Manzanares, A., Nomm, S., Bahsi, H., 2019b. In-depth feature selection and ranking for automated detection of mobile malware. In: ICISSP, pp. 274–283.

Guerra-Manzanares, A., Nõmm, S., Bahsi, H., 2019c. Time-frame analysis of system calls behavior in machine learning-based mobile malware detection. In: 2019 International Conference on Cyber Security for Emerging Technologies (CSET). IEEE, pp. 1–8.

Hei, Y., Yang, R., Peng, H., Wang, L., Xu, X., Liu, J., Liu, H., Xu, J., Sun, L., 2021. Hawk: rapid android malware detection through heterogeneous graph attention networks. IEEE Trans. Neural Netw. Learn. Syst. 1–15. doi:10.1109/TNNLS.2021.3105617.

Hou, S., Saas, A., Chen, L., Ye, Y., 2016. Deep4maldroid: a deep learning framework for android malware detection based on Linux kernel system call graphs. In: 2016 IEEE/WIC/ACM International Conference on Web Intelligence Workshops (WIW), pp. 104–111. doi:10.1109/WIW.2016.040.

Irolla, P., Dey, A., 2018. The duplication issue within the drebin dataset. J. Comput. Virol. Hack. Tech. 14 (3), 245–249.

Islam, Z., 2021. Android malware on the rise, google's os is more "interesting" to cybercriminals than apple IoS. https://www.techspot.com/news/91519-android-more-interesting-average-cybercriminal-than-ios-malware.html.

Isohara, T., Takemori, K., Kubota, A., 2011. Kernel-based behavior analysis for android malware detection. In: 2011 Seventh International Conference on Computational Intelligence and Security. IEEE, pp. 1011–1015.

Jaiswal, M., Malik, Y., Jaafar, F., 2018. Android gaming malware detection using system call analysis. In: 2018 6th International Symposium on Digital Forensic and Security (ISDFS), pp. 1–5. doi:10.1109/ISDFS.2018.8355360.

Jang, J.-w., Yun, J., Woo, J., Kim, H.K., 2014. Andro-profiler: anti-malware system based on behavior profiling of mobile malware. In: Proceedings of the 23rd International Conference on World Wide Web, pp. 737–738.

Jiang, X., Zhou, Y., 2013. Android Malware. Springer.

Johnson, J., 2021. Development of new android malware worldwide from june 2016 to march 2020. https://www.statista.com/statistics/680705/global-android-malware-volume/.

Jordaney, R., Sharad, K., Dash, S.K., Wang, Z., Papini, D., Nouretdinov, I., Cavallaro, L., 2017. Transcend: detecting concept drift in malware classification models. In: 26th {USENIX} Security Symposium ({USENIX} Security 17), pp. 625–642.

Kapratwar, A., Di Troia, F., Stamp, M., 2017. Static and dynamic analysis of android malware. In: ICISSP, pp. 653–662.

Kiss, N., Lalande, J.-F., Leslous, M., Tong, V.V.T., 2016. Kharon dataset: android malware under a microscope. In: The {LASER} Workshop: Learning from Authoritative Security Experiment Results ({LASER} 2016), pp. 1–12.

Lei, T., Qin, Z., Wang, Z., Li, Q., Ye, D., 2019. Evedroid: event-aware android malware detection against model degrading for IoT devices. IEEE Internet Things J. 6 (4), 6668–6680. doi:10.1109/JIOT.2019.2909745.

Lin, Y.-D., Lai, Y.-C., Chen, C.-H., Tsai, H.-C., 2013. Identifying android malicious repackaged applications by thread-grained system call sequences. Comput. Secur. 39, 340–350.

Lindorfer, M., Neugschwandtner, M., Platzer, C., 2015. Marvin: efficient and comprehensive mobile app classification through static and dynamic analysis. In: 2015 IEEE 39th Annual Computer Software and Applications Conference, vol. 2. IEEE, pp. 422–433.

Lipovsky, R., Stefanko, L., Branisa, G., 2017. Trends in android ransomware. https://www.welivesecurity.com/wp-content/uploads/2017/02/ESET_Trends_2017_in_Android_Ransomware.pdf.

Liu, Z., Wang, R., Japkowicz, N., Tang, D., Zhang, W., Zhao, J., 2021. Research on unsupervised feature learning for android malware detection based on restricted boltzmann machines. Future Gener. Comput. Syst. 120, 91–108.

Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., Zhang, G., 2018. Learning under concept drift: a review. IEEE Trans Knowl Data Eng 31 (12), 2346–2363.

du Luxembourg, U., 2021. Androzoo - lists of apks. https://androzoo.uni.lu/lists.

Mahindru, A., Sangal, A., 2021. MIdroid-framework for android malware detection using machine learning techniques. Neural Comput. Appl. 33 (10), 5183–5240.

Malik, S., Khatter, K., 2016. System call analysis of android malware families. Indian J. Sci. Technol. 9 (21), 1–13.

Margara, A., Rabl, T., 2018. Definition of Data Streams. Springer International Publishing, Cham, pp. 1–4.

McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupé, A., et al., 2017. Deep android malware detection. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp. 301–308.

Narayanan, A., Yang, L., Chen, L., Jinliang, L., 2016. Adaptive and scalable android malware detection through online learning. In: 2016 International Joint Conference on Neural Networks (IJCNN), pp. 2484–2491. doi:10.1109/IJCNN.2016.7727508.

Naval, S., Laxmi, V., Rajarajan, M., Gaur, M.S., Conti, M., 2015. Employing program semantics for malware detection. IEEE Trans. Inf. Forensics Secur. 10 (12), 2591–2604. doi:10.1109/TIFS.2015.2469253.

Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G., 2019. Mamadroid: detecting android malware by building Markov chains of behavioral models (extended version). ACM Trans. Privacy Secur. (TOPS) 22 (2), 1–34.

Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L., 2019. {TESSERACT}: eliminating experimental bias in malware classification across space and time. In: 28th {USENIX} Security Symposium ({USENIX} Security 19), pp. 729–746.

Rathore, H., Sahay, S.K., Nikam, P., Sewak, M., 2021. Robust android malware detection system against adversarial attacks using q-learning. Inf. Syst. Front. 23 (4), 867–882.
Saif, D., El-Gokhy, S., Sallam, E., 2018. Deep belief networks-based framework for malware detection in android systems. Alex. Eng. J. 57 (4), 4049–4057.
Samsung, 2021. About knox. https://www.samsungknox.com/en/about-knox.
Saracino, A., Sgandurra, D., Dini, G., Martinelli, F., 2018. Madam: effective and efficient behavior-based android malware detection and prevention. IEEE Trans. Dependable Secure Comput. 15 (1), 83–97. doi:10.1109/TDSC.2016.2536605.
Sasidharan, S.K., Thomas, C., 2021. Prodroidan android malware detection framework based on profile hidden Markov model. Pervasive Mob. Comput. 72, 101336.
Sharma, T., Rattan, D., 2021. Malicious application detection in android—A systematic literature review. Comput. Sci. Rev. 40, 100373.
Shipman, M., 2011. More bad news: two new pieces of android malware—plankton and yzhcsms. https://news.ncsu.edu/2011/06/wms-android-plankton/.
Sihag, V., Vardhan, M., Singh, P., Choudhary, G., Son, S., 2021. De-lady: deep learning based android malware detection using dynamic features. J. Internet Serv. Inf. Secur. (JISIS) 11 (2), 34–45.
Singh, L., Hofmann, M., 2017. Dynamic behavior analysis of android applications for malware detection. In: 2017 International Conference on Intelligent Communication and Computational Techniques (ICCT), pp. 1–7. doi:10.1109/INTELCCT.2017.8324010.
Statista, 2021. Mobile operating system market share worldwide, July 2020–July 2021. https://gs.statcounter.com/os-market-share/mobile/worldwide.
Surendran, R., Thomas, T., Emmanuel, S., 2020. A tan based hybrid model for android malware detection. J. Inf. Secur. Appl. 54, 102483.
Tchakounté, F., Dayang, P., 2013. System calls analysis of malwares on android. Int. J. Sci. Technol. 2 (9), 669–674.
Tong, F., Yan, Z., 2017. A hybrid approach of mobile malware detection in android. J. Parallel Distrib. Comput. 103, 22–31.
Vidal, J.M., Orozco, A.L.S., Villalba, L.G., 2017. Malware detection in mobile devices by analyzing sequences of system calls. World Acad. Sci., Eng. Technol., Int. J. Comput., Electr., Autom., Control Inf. Eng. 11 (5), 594–598.
Vinod, P., Zemmari, A., Conti, M., 2019. A machine learning based approach to detect malicious android apps using discriminant system calls. Future Gener. Comput. Syst. 94, 333–350.
Wahanggara, V., Prayudi, Y., 2015. Malware detection through call system on android smartphone using vector machine method. In: 2015 Fourth International Conference on Cyber Security, Cyber Warfare, and Digital Forensic (CyberSec). IEEE, pp. 62–67.
Wang, X., Li, C., 2021. Android malware detection through machine learning on kernel task structures. Neurocomputing 435, 126–150.
Wei, F., Li, Y., Roy, S., Ou, X., Zhou, W., 2017. Deep ground truth analysis of current android malware. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. Springer, pp. 252–276.
Xiao, X., Fu, P., Xiao, X., Jiang, Y., Li, Q., Lu, R., 2015. Two effective methods to detect mobile malware. In: 2015 4th International Conference on Computer Science and Network Technology (ICCSNT), vol. 1. IEEE, pp. 1041–1045.
Xiao, X., Xiao, X., Jiang, Y., Liu, X., Ye, R., 2016. Identifying android malware with system call co-occurrence matrices. Trans. Emerg. Telecommun. Technol. 27 (5), 675–684.
Xiao, X., Zhang, S., Mercaldo, F., Hu, G., Sangaiah, A.K., 2019. Android malware detection based on system call sequences and LSTM. Multimed. Tools Appl. 78 (4), 3979–3999.
Xu, K., Li, Y., Deng, R., Chen, K., Xu, J., 2019. Droidevolver: self-evolving android malware detection system. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, pp. 47–62.
Yu, R., 2013. Ginmaster : a case study in android malware. https://www.virusbulletin.com/conference/vb2013/abstracts/ginmaster-case-study-android-malware.
Yu, W., Zhang, H., Ge, L., Hardy, R., 2013. On behavior-based detection of malware on android platform. In: 2013 IEEE Global Communications Conference (GLOBECOM), pp. 814–819. doi:10.1109/GLOCOM.2013.6831173.
Yuan, Z., Lu, Y., Wang, Z., Xue, Y., 2014. Droid-sec: deep learning in android malware detection. In: Proceedings of the 2014 ACM conference on SIGCOMM, pp. 371–372.
Zhang, X., Zhang, Y., Zhong, M., Ding, D., Cao, Y., Zhang, Y., Zhang, M., Yang, M., 2020. Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp. 757–770.
Zhou, Y., Jiang, X., 2012. Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security and Privacy, pp. 95–109. doi:10.1109/SP.2012.16.
Zyblewski, P., Sabourin, R., Woźniak, M., 2021. Preprocessed dynamic classifier ensemble selection for highly imbalanced drifted data streams. Inf. Fusion 66 (June 2020), 138–154. doi:10.1016/j.inffus.2020.09.004.

Alejandro Guerra Manzanares is a Ph.D. candidate at the Center for Digital Forensics and Cyber Security, Department of Software Science, Tallinn University of Technology (Estonia). He received a BA degree in criminology from the Autonomous University of Barcelona (Spain) in 2013, and a BS degree in ICT engineering from the Polytechnic University of Catalonia (Spain) in 2017. In 2018, he received a M.Sc. in cyber security from Tallinn University of Technology (Estonia). His research interests are in the application of machine learning techniques to digital forensics and cybersecurity-related issues, such as mobile malware detection and IoT botnet detection.

Hayretdin Bahsi is a research professor at the Center for Digital Forensics and Cyber Security at Tallinn University of Technology, Estonia. He has two decades of professional and academic experience in cybersecurity. He received his Ph.D. from Sabanc University (Turkey) in 2010. He was involved in many R&D and consultancy projects about cybersecurity as a researcher, consultant, trainer, project manager, and program coordinator at the National Cyber Security Research Institute of Turkey between 2000 and 2014. His research interests include machine learning and its application to cyber security and digital forensic problems.

Appendix 10

Publication X
A. Guerra-Manzanares and H. Bahsi. On the application of active learning to handle data evolution in android malware detection. *Conference paper*, *under review*, 2022

On the application of active learning to handle data evolution in Android malware detection

Alejandro Guerra-Manzanares and Hayretdin Bahsi

Tallinn University of Technology, Tallinn, Estonia

Abstract. Mobile malware detection remains a significant challenge in the rapidly evolving cyber threat landscape. Although the research about the application of machine learning methods to this problem has provided promising results, still, maintaining continued success at detecting malware in operational environments depends on holistically solving challenges regarding the feature variations of malware apps that occur over time and the high costs associated with data labeling. The present study explores the adaptation of the active learning approach for inducing detection models in a non-stationary setting and shows that this approach provides high detection performance with a minimal set of labeled data for a long time when the uncertainty-based sampling strategy is applied. The models that are induced using dynamic, static and hybrid features of mobile malware are compared against baseline approaches. Although active learning has been adapted to many problem domains, it has not been explored in mobile malware detection extensively, especially for non-stationary settings.

Keywords: mobile malware · Android · malware detection · active learning · concept drift · data evolution

1 Introduction

Mobile devices play a significant role in our personal and professional lives. Malicious actors target these devices for various purposes ranging from pursuing economic benefits to collecting information for espionage activities. Mobile malware is one of the greatest cyber threats in this digital ecosystem. Android is the most targeted mobile operating system (OS) by attackers, its share in the threat landscape constitutes 98% of the mobile cyber attacks [8]. The security efforts of Google and device vendors in this regard [14, 4] have not been able to put a stop to the increasing trend of this type of cyber attack. Machine learning methods have been proposed to detect malware [18] as these techniques may discriminate behavioral patterns of mobile apps to identify new malicious applications.

The research studies that apply machine learning methods to cyber security problems, in general, and mobile malware detection, in particular, usually validate their results on static data sets belonging to specific time frames (e.g., Drebin [2]). However, the threat landscape is subject to constant evolution due to the inherent attack-defense confrontation between the malicious actors and

the security experts in the domain. Relevant dynamic and static features of mobile malware have been proved to continuously change (i.e., legitimate apps are also prone to change to some extent) so that the discrimination capabilities of the learning models diminish over time [7]. Thus, handling the non-stationary property of the data should be one of the building blocks of an operational system to maintain continuous high detection performance for malware detection purposes. This denotes that the learning model should be retrained when a significant data distribution shift is detected. Based on the resources available, one option would be to perform periodic retraining of the model to guarantee an updated model regardless of the variations in the data.

Finding labeled data is always a significant challenge in the cyber security domain due to the lack of human resources or confidentiality concerns that eliminate the possibility of data sharing between different organizations. Although one-class models, which are trained on only legitimate samples, provide a solution to some extent, their performance is usually lower when compared to supervised models and they do not enable the induction of multi-class models which may be highly beneficial to identify malware families. Additionally, they require additional mechanisms to prove that legitimate samples are free from any malicious content, which refers to another form of labeling. Similar to all settings, the effectiveness of non-stationary ones also hugely depends on feeding the training sets with recent samples which are correctly labeled. Therefore, the design considerations of such operational detection systems should holistically address labeling and retraining aspects.

On the other side, despite the aforementioned problems, it does not mean that the cyber security vendors (i.e., companies providing mobile malware scanners or protection products in our case) cannot assign any resources for labeling. A typical vendor has teams of malware analysts that work on a daily basis to investigate the new malware samples. Therefore, we contemplate that both data labeling and model retraining can be achieved by adapting active learning to a non-stationary environment. Active learning approaches create an interactive channel between experts and models so that the models themselves select the most informative samples from an unlabeled data pool, ask the class label from the experts and incorporate their answers into the model. Active learning approaches aim to minimize the labeling efforts to achieve the highest model performance possible. These approaches are very instrumental in cases where obtaining unlabeled data is easy and cheap but labeling is expensive, which reflects the needs of our target problem [15].

In this study, we adapted a pool-based active learning approach that uses the uncertainty sampling strategy to a non-stationary setting and demonstrated its effectiveness in terms of detection performance and the required labeling effort for mobile malware detection in Android devices. We utilized the Android malware data set *KronoDroid* [5] which suits well for non-stationary model experiments as it has timestamped malware and *goodware* samples encompassing the whole Android historical timeline. Dynamic and static features of Android apps, more specifically, system calls and permissions, are used for inducing the

models separately or in hybrid mode. We compared our results with two baseline models: (1) a *batch retraining* strategy that uses all the previous labeled samples for inducing a model for the next time period, and (2) an iterative learning strategy that randomly selects a sample from the unlabeled data pool without using any informativeness criteria. Our results show that the uncertainty sampling method achieves over 91% F_1 score, on average, throughout a 7-year-long period while enormously minimizing the required labeling effort (i.e., 2-3% of the labeled samples are enough for high detection performance when compared to the batch retraining strategy).

The performance of active learning in non-stationary settings, subject to concept drift issues, has not been demonstrated comprehensively for mobile malware detection. Therefore, this study provides a solid contribution by proposing active learning for addressing the problems of such detection systems in operational settings.

It is important to note that although this study presents experimental results of a solution that covers both data labeling and model retraining, our focus was to elaborate on the trade-off between the labeling effort and its impact on model performance. We assumed that the model is retrained in fixed intervals. It is evident that the detection performance and labeling resource consumption may be also enhanced by different retraining approaches or intervals which can be coupled with various concept drift detectors. However, the detailed analysis of retraining options is out of scope in the present paper.

This paper is structured as follows. Section 2 provides background information and a summary of the related literature. Section 3 provides the methodology while Section 4 reports and discusses the main results. Section 5 concludes the study.

2 Background Information & Literature review

2.1 Background information

Active Learning A form of *semi-supervised learning* based on the assumption that a machine learning (ML) algorithm can yield better performance with fewer training iterations (i.e., less data) if it is allowed to select the data from which it learns [16]. For this purpose, a supervised model is trained with a small quantity of data (i.e., *active learner*) and enabled to submit *queries* for selected unlabeled data samples to a labeling *oracle* (i.e., a human expert). The main aim is to achieve high performance using as few labeled samples as possible, thus minimizing the cost of the data labeling process.

The selection of the specific instance for labeling (i.e., query instance) at each training iteration is based on an *informativeness* assessment of the whole set of unlabeled instances performed by the active learner using a specific query strategy [17]. The pool-based framework, described graphically in Figure 1, is the most common active learning approach. This approach assumes the existence of a small labeled data set and the availability of a large *pool* of unlabeled data

[16]. Query instances are selected from the unlabeled pool for expert annotation. The labeled data sample is then incorporated into the labeled training set which is used to update the knowledge of the ML model (i.e., retraining).

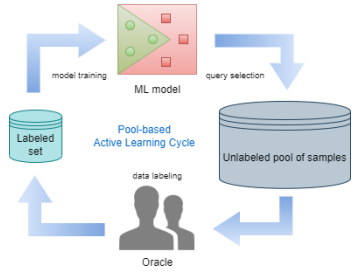


Fig. 1. Pool-based active learning framework

Various query strategies can be used to choose the *most informative* instance [17]. The most commonly used approach is *uncertainty sampling*, where the query instance is selected based on how certain is the learner about the class of the samples. In the *classification uncertainty* scoring strategy, the learner selects the instance (x) for which it is least certain about how to label (i.e., greatest uncertainty). More precisely, the strategy is based on the *least confidence* score (U) computed as:

$$U(x) = 1 - P(y^*|x) \tag{1}$$

where x is a specific instance and y^* is the most likely prediction for that instance.

Concept Drift Most machine learning models are built on the assumption of *stationary* data, so the testing data attributes are assumed to be similar to the training data attributes, not changing over time. However, in some problem domains, this assumption does not hold as the incoming data features distribution may change over time, thus affecting the *generalization* of the model against new data and, consequently, harming the effectiveness of the detection model over time, a phenomenon called *concept drift* [10]. More precisely, concept drift is observed when the underlying data distribution changes over time (i.e., $P_t(X) \neq P_{t+1}(X)$, where t and $t+1$ indicate consecutive non-overlapping periods) and affects the decision boundary of the classifier which impacts the target class estimation (i.e., $P_t(y|X) \neq P_{t+1}(y|X)$) and results in a decay of performance over time. The appearance of any type of concept drift requires the update of the knowledge of the learning model to the new data distribution to keep high-performance metrics [10].

As a result of the constant evolution of the threat landscape (e.g., new malware) and the inherent evolution of the Android framework (e.g., system up-

dates), Android malware detection systems are prone to concept drift issues. Therefore, an effective malware detection system must update its knowledge to sustain high detection performance over time.

2.2 Literature review

The first active learning applications in the cyber security domain were carried out for network intrusion detection using the widely-known KDD Cup 1999 data set [1, 9]. Uncertainty sampling strategy achieved the reduction of required labeled data by a factor of eight when compared to the baseline strategy, random sampling, [1] whereas confidence measures identified by transductive reliability estimation reduced this factor to the fifth of the same baseline model [9].

Malicious *doc* files were detected by an active learning solution that yielded a high-performance model with 14% of the labeled samples used by the passive learning model [11]. The directory paths that are retrieved from the hierarchical structure of office documents constitute the features of the detection model in the corresponding study. Uncertainty sampling is complemented by a rare-class detection that is applied in the form of multi-class formulation to annotate malware families [3]. The main idea is to include representative samples from all families while selecting the samples from the pool. The proposed approach was applied to two problems, detection of malicious pdf files and network attacks.

A few Android research studies have concentrated on concept drift handling and none of them used the active learning approach in their method. MaMaDroid [12] and DroidEvolver [19] used API calls and traditional ML and online algorithms, respectively, to handle concept drift, whereas [6] used system calls and a data stream methodology to tackle the issue.

[13] draws the attention to experimental biases in malware detection research including the temporal bias and demonstrates how validation designs with such biases influence the results obtained. Although it also demonstrates some results regarding the active learning application in non-stationary settings, the purpose is to underline the biases rather than elaborating on an active learning approach. In our study, we investigate the impact of feature types (i.e., dynamic, static, or both) and data balancing strategies on the detection results. We used a data set that encompasses a longer time frame (i.e., our experimentation covers a period of seven years, whereas [13] covers two years of Android data). Our benchmark includes a comparison with random sampling to show the effectiveness of the uncertainty sampling strategy in our problem formulation.

Similarly, active learning is used as one of the methods for maintaining the detection stability of a mobile malware detection model over time in [20]. Although this period encompasses a long period, five years, this study concentrates on the representation of the feature space rather than the comprehensive evaluation of the active learning approach. More specifically, this study proposes an abstract representation of API call features to grasp better semantic similarities between different malware samples, thus, the model induced using those features can detect the malware evolution better.

3 Methodology

The following sections describe the data set used in our experimental setup, the methodological workflow and the tested scenarios to handle concept drift for Android malware detection using active learning techniques.

3.1 Data set and data features

The data set used in this research is *KronoDroid* [5], a hybrid-featured, labeled, and fully timestamped Android data set, which makes it the most suitable data set for Android concept drift exploration among the available data sets for the purpose of Android malware detection. The *real device* data set was used due to its larger size (i.e., 41,382 malware samples, and 36,755 benign apps). For this study, system calls and permissions features were used as models' input features, along with the *first seen* timestamp and the class labels, which were used to order the samples along the Android historical timeline and class identification, respectively. The *first seen* timestamp, retrieved from *VirusTotal*, provides information about when the sample was received for the first time (i.e., user submission) by the detection system. The usage of this timestamp enabled us to simulate the constant data stream of Android data samples as a realistic scenario for a malware scanner company dealing with an Android malware detection system subject to concept drift issues. For model induction, three sets of input features were used to describe the apps, namely, static (permissions), dynamic (system calls), and hybrid (system calls and permissions) with lengths 166, 288, and 454, respectively, and composed of different variable types. Table 1 summarizes the data set used in this study.

Table 1. Data set summary

Data	Size	Description
Benign samples	36755	Time frame: 2008-2020
Malware samples	41382	Time frame: 2008-2020
Permissions	166	Binary features
System calls	288	Numeric features
Hybrid (perms + syscalls)	454	Binary and numeric features

3.2 Workflow and scenarios

To explore the application of active learning for concept drift handling and adaptation, the data was divided into consecutive data chunks, simulating a data stream covering the Android historical timeline. The samples were ordered and grouped in data chunks, according to their timestamp. Additionally, maximum

data chunk size and time constraints were used to ensure the existence of sufficient data (i.e., over 100 samples) for every chunk in the whole time frame analyzed. The same classifier algorithm was used in all scenarios and was retrained using different concept drift-handling strategies.

The test scenarios for concept drift handling are described as follows:

- *Batch retraining*: This strategy updates the detection model by retraining the classifier using the whole amount of data available in each specific chunk, and the retrained model is used to forecast the labels for each subsequent period. Therefore, at time t all data from previous time periods (i.e., $s_0, .., s_{t-1}$, where s identifies a data set belonging to a specific time period, t) was used to update the model, and forecast the labels of s_{t+1}. Next, the whole data set belonging to $t + 1$ was used to update the model (i.e., retraining) and forecast labels for s_{t+2}. This cycle was repeated for each data chunk until the end of the analysis period. This batch-retraining approach is the frequent solution utilized for concept drift adaptation. It was used as a baseline in our experimentation.
- *Active learning*: This strategy updated the detection model by selecting the *most informative* instances for each data chunk, one at a time, until a predefined performance threshold was reached. The *classification uncertainty* score, as described in Section 2.1, was used to rank and select one instance at a time from the unlabeled pool of instances (i.e., whole data chunk). The selected instance was labeled by the *oracle* and used to retrain the model. The rest of the data chunk was used to evaluate the performance increase/decrease after the single retraining step. The training cycle, as depicted in Figure 1, was repeated until a performance score threshold was achieved. The remaining data, not used in the iterative training steps, were discarded and the trained model was used to forecast all the samples for the next period, as in *batch retraining*. If the performance retrieved processing all the data chunk was lower than the established threshold, the model was rolled back to its best performer configuration and used to forecast the subsequent period data.
- *Random sample selection retraining*: This strategy uses the same iterative training steps as the active learning approach but, in this case, no score is used to select the most informative instances from the unlabeled pool (i.e., whole data chunk). Instead, random sample selection is used. This strategy enabled us to simulate the scenario where a bunch of unlabeled data is available, but no specific criterion is used to select the instances. Thus, samples are selected at random. This model provides the baseline to assess the effectiveness of the sample selection strategy in terms of data labeling minimization.

All the testing scenarios were performed using the same classification algorithm, induced separately using the three feature sets (i.e., static, dynamic, and hybrid). The performance of the induced models, using the different sets of features for all the strategies, was retrieved and compared. In all cases, the

model trained using data from period $t-1$ was used to forecast the labels of the data belonging to the subsequent period, s_t. The main difference among the approaches lies in the training data used and, more specifically, in the strategy used to select the samples for model updating (i.e., all data, random selection or uncertainty score).

The performance of the detection models using the described retraining strategies to handle concept drift was evaluated using two relevant binary classification performance metrics: *accuracy* and F_1 *score* metrics. These metrics were retrieved for each data chunk (i.e., period).

The accuracy metric reports the number of correctly predicted data points out of all the test data points, whereas the F_1 *score* metric is the harmonic mean of *precision* and *recall*. The *precision* of a classification model informs about the fraction of true positive (i.e., malware) data points that the model correctly classified as positive (i.e., malware), while *recall* reports the fraction of samples classified as positive among the total number of positive samples in the testing set.

4 Results & Discussion

The three concept drift-handling retraining strategies described were evaluated using the same base classifier, a Random Forest instance trained using the same initial data set and the default values of Python's *scikit-learn* library. The initial training data set encompassed the months of July and August 2011. This period was selected as it provided enough data to generate a good initial base classification model. Despite that, as the initial training data set was not balanced, a data balancing technique was applied. Two data balancing methods were used (i.e., *random undersampling* and *random oversampling*) and their impact was evaluated. As explained, the remaining data, ordered by their timestamp, were split into consecutive data chunks using temporal and size constraints. Based on experimental tests, the maximum temporal constraint or time window was set at 60 days (i.e., $\approx$ 2 months) and the maximum data pool size set to 4000. Therefore, the maximum data chunk size was composed of 4000 data samples, spanning a maximum of 60 days of data per chunk. The time period analyzed ranges from the initial time frame (i.e., July-August 2011) to May-June 2018. Posterior time frames did not provide enough quantity of data to continue our experimentation (e.g., chunks with less than 50 samples), so the experimentation time frame and the provided results encompass 7 years of the Android history.

The active learning query strategy was implemented using Python's *modal* library, while the balancing techniques used the *imblearn* library. Given the inherent randomness of some of the strategies (i.e., random selection) and techniques used (i.e., random under/oversampling) each of the evaluated scenarios was repeated 30 times and the average values were reported. The performance threshold to stop processing data for the active learning approaches was set at 0.95 F_1 score. Therefore, if after processing n samples, an F_1 performance of 0.95 (out of 1) or higher was obtained, no more data was labeled in that quarter

and the resulting model was used to forecast the labels for the next period data. When the performance threshold was not achieved, the highest F_1 performer model was used.

Table 2 provides the obtained results using all the described concept drift-handling approaches. More specifically, the column *feature set* describes the input features used by each specific model tested and the *balancing method* column reports the technique used to balance the initial data set, in the case of the two query strategies used (i.e., random and uncertainty), as well as all data chunks for the batch approach (i.e., to avoid that the imbalanced data chunks generated biased RF models). For each combination of the feature sets and balancing methods, three strategies to handle concept drift were evaluated, described in Section 3.2, and referenced in the *query strategy* column. The remaining columns in Table 2 report the performance metrics that enabled us to analyze and compare all the evaluated approaches. The *labeled samples* column informs about the average number of samples processed by each model (i.e., , $\overline{x}$), that is, the number of instances labeled, to reach the performance threshold, $F_1 \geq 95\%$. The columns F_1 *score* and *accuracy* provide the average performance of the trained models in all time windows in the analyzed time frame (e.g., 45 data chunks spanning between September/October 2011 and May/June 2018). The reported values for labeled samples and the performance metrics are the average values of the 30 tests performed for each specific scenario. The standard deviation (i.e., s) is reported to contextualize better the mean value as a data descriptor. Additionally, for the *labeled samples*, the proportion of the average number of labeled samples reported in relation to the total data available in the analyzed period is reflected by the % column.

As can be observed in Table 2, when the permissions features are used, the active learning approach (i.e., uncertainty) provides similar performance as the baseline model (i.e., batch, using all data), but requires the smallest number of data samples among the tested strategies. The uncertainty-based active learning approach minimizes the data labeling needed to achieve similar performance as the other two approaches using either of the balancing techniques. More precisely, the batch approach, which requires the labeling of all the data samples, shows slightly better performance than the active learning approaches, but these show significantly lower data labeling requirements. In this regard, the uncertainty-based active learning approach outperforms the random selection approach by using less than 18% of the total data in both cases. Even though both single query-based retraining approaches show benefits over the batch approach, the active learning approach requires three times fewer data than the random instance selection to achieve the same performance metrics. This fact evidences that, in the permissions case, the single query-based gradual modification of the classifier decision boundary shows benefits when it is compared to the baseline model, which uses batch processing (all data). The random approach shows slightly lower performance than the baseline model, but with less data labeling needs, evidencing that more data might not be necessary to handle concept drift effectively but that, more importantly, the instance-based gradual retraining of

Table 2. Testing scenarios results

Feature set	Balancing method	Query strategy	Labeled samples			F_1 score		Accuracy	
			$\overline{x}$	s	%	$\overline{x}$	s	$\overline{x}$	s
Permissions	Oversampling	Batch	67068	0	100	91.2	0.4	92.5	0.4
		Random	30100.4	129.8	44.9	89.4	1.0	90.3	1.0
		Uncertainty	11845.6	41.7	17.7	89.4	0.6	90.4	0.6
	Undersampling	Batch	67068	0	100	90.9	0.8	92.4	0.8
		Random	29409.9	113.5	43.9	89.5	1.1	90.3	1.1
		Uncertainty	9281.4	35.5	13.8	89.6	0.7	90.5	0.6
Syscalls	Oversampling	Batch	67068	0	100	85.1	0.8	86.1	0.7
		Random	45028.9	127.8	67.1	84.1	0.9	84.9	0.9
		Uncertainty	13098.8	38.6	19.5	84.5	0.9	85.3	0.8
	Undersampling	Batch	67068	0	100	82.7	1.2	83.3	1.2
		Random	45378.7	118.5	67.7	84.5	0.9	85.0	1.0
		Uncertainty	12748.3	52.3	19.0	85.1	1.0	85.5	1.0
Hybrid	Oversampling	Batch	67068	0	100	92.8	0.5	93.5	0.4
		Random	22057.2	121.5	32.9	90.9	1.0	91.2	1.0
		Uncertainty	1991.9	8.9	3.0	91.6	1.2	91.9	1.2
	Undersampling	Batch	67068	0	100	92.5	0.6	93.1	0.6
		Random	20978.4	116.1	31.3	91.0	1.1	91.1	1.1
		Uncertainty	1459.4	6.3	2.2	91.7	1.4	91.9	1.4

the model may be more beneficial to handle concept drift effectively. There are no major differences in performance in any cases when both balancing methods are compared. However, the *undersampling* approach provides similar performance metrics to the *oversampling* method with significantly fewer data in the *active learning* case (i.e., 28% more data is needed, on average, for the oversampling case than for the undersampling scenario). In conclusion, the best results in terms of both performance and minimization of data labeling needs, when the permissions feature set is used, are obtained using the undersampling balancing strategy combined with the uncertainty-based active learning query approach.

Comparatively, the system calls feature set produced the worst performance models among all tested models in both evaluated metrics, the number of labeled samples needed and performance achieved. More precisely, the batch strategy using undersampling provides average performance metrics below 85%, which are slightly better when oversampling is used. These performance metrics are the worst across all models and feature sets, as none of the tests using the permissions or the hybrid feature sets go lower than 89.4% F_1 and 90.3% accuracy. However, the system calls-based model performance is significantly improved when a single query strategy is implemented, and, more specifically, when the uncertainty-based active learning approach is used, reaching similar performance as the baseline model, as in the permissions case, and even outperforming when undersampling is used. Despite that, the labeling needs for the uncertainty-based active learning, which, again, minimizes the labeling cost, is superior to

the permissions case for both sampling techniques (i.e., a minimum of 19% of the data has to be labeled by the oracle). It is worth noting that random selection reaches similar performance as the uncertainty-based strategy but requires, in both cases, over 66.7% of the whole data to be labeled. Thus, again, the single query approach shows advantages over batch processing to handle concept drift effectively, especially when the uncertainty-based active learning approach is applied.

The hybrid feature sets, which combine the permissions and system calls sets for model induction, provide the best overall models, in all cases. The active learning approach using the uncertainty criterion reaches a slightly lower performance than the baseline performance implementing the batch approach. However, in this case, the benefits of the active learning approach are especially evident for both balancing techniques. The labeling needs are significantly lowered, not over-passing 9% of the whole data set. As a result, they provide the best performance-labeling trade-off results among all the test scenarios. In this regard, the best model of all the tested scenarios is obtained using the active learning approach combined with undersampling, yielding an average 91.7% F_1 score and 91.6% accuracy using, on average, only 1460 samples (i.e., $\approx$ 2.2% of the total data) to provide effective detection in the seven-year-long study period. Comparatively, the uncertainty-based active learning approach for the hybrid-featured models requires 10-15 times fewer data than the random query approach to achieve better performance results and 50 times fewer data to reach similar detection performance than the baseline models. These results show that the hybrid feature set generates better discriminatory models which benefit notably from the active learning approach, being able to handle concept drift with a significantly reduced quantity of labeled data belonging to specific time frames along a seven-year-long time period (i.e., from September-October 2011 to May-June 2018).

To further explore the results, the summary values reported in Table 2 are provided in more fine-grained detail on the analyzed historical timeline of Android in Fig. 2, 3, 4, and 5. More specifically, in these figures, the X-axis reports the time frame of the specific data chunk, encompassing, at maximum, two months of data. The axis labels provide the year and month separated by a slash (e.g., 2011/9-10 reports data comprised between September and October 2011). The left Y-axis reports the number of samples included in every data chunk (i.e., grey color), thus composing the unlabeled pool of samples for the active learning approaches, that were actually labeled by the oracle (i.e., blue color). Given the degree of randomness of the approaches used, the reported values for the number of labeled samples (i.e., blue area on the bars) are mean values with the confidence interval of the mean estimation reported by the white whiskers that extend over and below the mean (i.e., confidence level 95%). The average performance scores obtained on each data chunk are reported by the yellow (i.e., accuracy) and blue (i.e., F_1 score) lines placed on top of the bar chart, and ranging from 0 to 1 (i.e., right Y-axis). The standard deviation of these performance metrics is provided by the colored ribbons surrounding the average lines. Fig. 2

provides the average results for the uncertainty-based active learning approach when undersampling and the permissions set were used. Fig. 3 reports the same information when the system calls set is used while Fig. 4 provides the hybrid feature set-related information. These figures enable us to compare the impact of the feature set under the same conditions (i.e., uncertainty-based active learning approach using undersampling). Lastly, Fig. 5 enables the comparison between the best active learning model (i.e., hybrid feature set, undersampling using uncertainty score, as depicted in Fig. 4) and the random query strategy for the same feature set and sampling approach configuration.

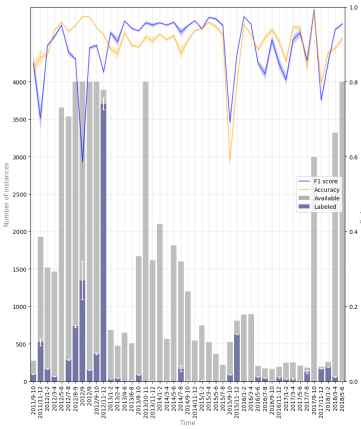


Fig. 2. Permissions, undersampling, and uncertainty-based model results

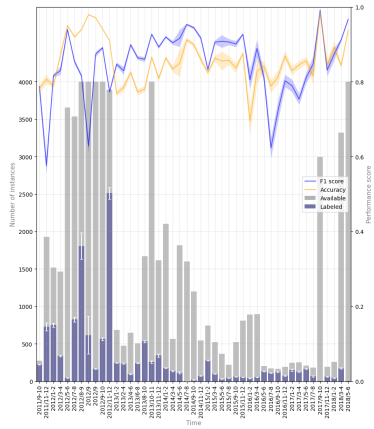


Fig. 3. System calls, undersampling, and uncertainty-based model results

As can be observed, in Figure 2, the permissions feature set enabled the handling of concept drift using significantly less labeled data than the system calls feature set, depicted in Figure 3. With some minor exceptions (e.g., 11-12/2012), the permissions feature set required fewer labeled data per chunk to sustain the training target of 95% F_1 score, over-passing this score in many chunks, thus no data was labeled for training purposes (e.g., 10-11/2013, 11-12/2013, 1-2/2014, 3-4/2014, and 5-6/2014). Despite the goodness shown by the permissions feature set to handle concept drift using the active learning approach, these results are significantly outperformed by the hybrid feature set, which combines the system calls and permissions feature set. In this case, a reduced proportion of the chunk data is labeled in every chunk to achieve high-performance metrics (e.g., 9-10/2011, 11-12/2011) with extended periods of almost no training data needs (e.g., from 4-6/2013 to 7-8/2015). Therefore, the high-dimensional feature space generated by the joint usage of both feature sets enabled the handling of con-

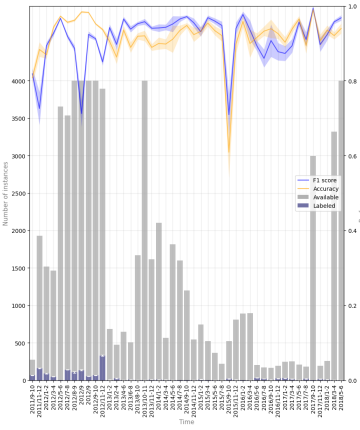


Fig. 4. Hybrid, undersampling, and uncertainty-based model results

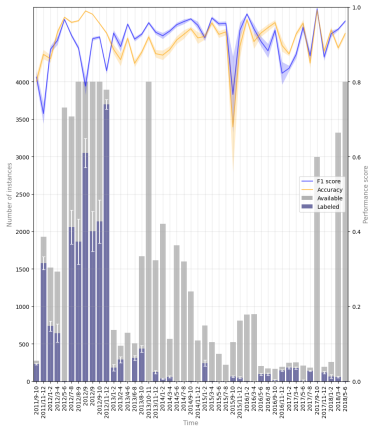


Fig. 5. Hybrid, undersampling, and random selection model results

cept drifts better than any other approach, keeping high-performance metrics with just a few samples labeled per chunk. Even though this feature set reduces the data needs in all approaches and strategies, the uncertainty-based query strategy shows significant improvement concerning random query selection, as can be seen in Fig. 5. The random query strategy requires significantly more labeled data per data chunk to sustain performance and address concept drift, evidencing the superiority of the uncertainty-based selection over random query selection.

The obtained results show that the active learning approach, in its most basic form (i.e., uncertainty sampling) can be effectively used to handle concept drift, keeping high-performance metrics while minimizing the data labeling efforts (i.e., the amount of labeled data needed to keep high performance). As a result, active learning might be an efficient and effective solution to handle concept drift in environments where a large quantity of unlabeled data is available but with high labeling costs. It allows focusing the labeling effort on the *relevant* data to improve the model and discard the *irrelevant* data samples that may not provide benefit to the model. Despite that, uncertainty sampling may yield *biased* classifiers and sub-optimal models if the initial data set is too small or not representative as the model *certainty* is used to rank the *informativeness* or relevancy of the samples. To avoid that, other query strategies could be used such as query by committee or ranked batch-mode [16]. In our case, the initial data set has been proved reliable and large enough to overcome this issue and the performance obtained by the models does not change significantly (i.e.,

standard deviation values are not large). The exploration of the benefits of other approaches constitutes part of our future work.

The comparative performance metrics provided in this study show that the gradual modification of the decision boundary caused by the addition of a single relevant sample in the training data set provides high-performance models using significantly less data than the batch retraining approach. Random instance selection improves the labeling needs concerning the batch retraining approach, but they are both outperformed significantly by the active learning approach. Random selection requires consistently more data to achieve roughly the same (but not better) performance metrics than the uncertainty sampling approach. This fact evidences the goodness of the active learning approach to induce great performance models with significantly fewer data needs.

To address imbalance issues, two balancing techniques were explored. Random oversampling balances the data by generating artificial but similar data points for the underrepresented class, whereas random oversampling selects a random sample from the overrepresented class to match the number of samples in the underrepresented class. Even though both approaches worked similarly for random and batch strategies, the undersampling approach provided distinctive benefits using the active learning approach for the permissions and hybrid feature sets. This technique minimized the data labeling efforts significantly while producing great discriminatory results. The exploration of more complex balancing approaches is part of our future work.

This paper explores the application of *active learning* as an alternative approach to deal with concept drift in Android malware detection. The related methods in the literature [19, 12, 20, 13] propose the usage of more complex algorithmic solutions that require extensive data labeling efforts and intensive computational resources. The active learning approach, due to its focus on data labeling minimization, reduces the computational load and resources needed to deal effectively with concept drift issues, as demonstrated in this paper. More specifically, the comparison of the results obtained in this study with related works [6, 19, 12, 20, 13] evidences the goodness of the active learning approach to maximize detection performance metrics while minimizing labeling needs and, consequently, computational resources. Most of these proposed solutions assume the labeling of the whole data set at each training step thus they are analogous to the batch retraining approach, which was used as a baseline in our study. Besides, the detection solutions in the literature are more computationally intensive due to their algorithmic complexity. For instance, some methods combine the output of a pool of classifiers [6, 19] or use complex adaptive pipelines [12] that increase the burden of system maintenance and the overall resources needed to operate and update the detection system. In our benchmarking, a single classifier model, based on a traditional machine learning algorithm (i.e., Random Forest), using the *active learning* query strategy was capable of providing high detection performance for a long period (i.e., from 2011 to 2018) with few data updates over time and many time frames with zero labeling needs. The performance results are similar to the ones proposed by [6] and the baseline approach

(i.e., batch retraining), but use a less complex system, which is easier to maintain and requires less computational and labeling resources. It also outperforms the rest of concept drift-handling methods in the related literature, which manifest significant performance decay over time [19, 12].

5 Conclusions

The active learning approach is built on the assumption that a machine learning model can learn faster (i.e., in fewer training steps) and with less data if the model is allowed to select the data from which it learns. This approach combines the knowledge of an *oracle* and instance selection by the supervised model to enhance performance and minimize data needs. To the best of our knowledge, this is the first study that leverages *active learning* to handle concept drift in Android malware detection. Our results show that the active learning approach, in its most basic form, allows effective concept drift handling in Android malware detection and, more interestingly, minimizes the data labeling needs. Consequently, it becomes an option worth considering for enhancing the ML-based detection systems in cyber security environments (e.g., malware protection companies, security operating centers dealing with Android malware detection), where a large body of unlabeled data is constantly available but the high labeling cost associated makes the task infeasible and prohibitive, thus affecting the detection capabilities of the system.

References

1. Almgren, M., Jonsson, E.: Using active learning in intrusion detection. In: Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004. pp. 88–98. IEEE (2004)
2. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., Siemens, C.: Drebin: Effective and explainable detection of android malware in your pocket. In: Ndss. vol. 14, pp. 23–26 (2014)
3. Beaugnon, A., Chifflier, P., Bach, F.: Ilab: An interactive labelling strategy for intrusion detection. In: International Symposium on Research in Attacks, Intrusions, and Defenses. pp. 120–140. Springer (2017)
4. Google: Google play protect. https://developers.google.com/android/play-protect (2021)
5. Guerra-Manzanares, A., Bahsi, H., Nõmm, S.: Kronodroid: Time-based hybrid-featured dataset for effective android malware detection and characterization. Computers & Security **110**, 102399 (2021)
6. Guerra-Manzanares, A., Luckner, M., Bahsi, H.: Android malware concept drift using system calls: Detection, characterization and challenges. Expert Systems with Applications p. 117200 (2022). https://doi.org/https://doi.org/10.1016/j.eswa.2022.117200
7. Guerra-Manzanares, A., Nomm, S., Bahsi, H.: In-depth feature selection and ranking for automated detection of mobile malware. In: ICISSP. pp. 274–283 (2019)

8. Kaspersky: Mobile security: Android vs ios - which one is safer? https://www.kaspersky.com/resource-center/threats/android-vs-iphone-mobile-security (2020)
9. Li, Y., Guo, L.: An active learning based tcm-knn algorithm for supervised network intrusion detection. Computers & security **26**(7-8), 459–467 (2007)
10. Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., Zhang, G.: Learning under concept drift: A review. IEEE Transactions on Knowledge and Data Engineering **31**(12), 2346–2363 (2018)
11. Nissim, N., Cohen, A., Elovici, Y.: Aldocx: detection of unknown malicious microsoft office documents using designated active learning methods based on new structural feature extraction methodology. IEEE Transactions on Information Forensics and Security **12**(3), 631–646 (2016)
12. Onwuzurike, L., Mariconti, E., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G.: Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version). ACM Trans. Priv. Secur. **22**(2) (apr 2019). https://doi.org/10.1145/3313391, https://doi.org/10.1145/3313391
13. Pendlebury, F., Pierazzi, F., Jordaney, R., Kinder, J., Cavallaro, L.: {TESSERACT}: Eliminating experimental bias in malware classification across space and time. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 729–746 (2019)
14. Samsung: About knox. https://www.samsungknox.com/en/about-knox (2021)
15. Schütze, H., Velipasaoglu, E., Pedersen, J.O.: Performance thresholding in practical text classification. In: Proceedings of the 15th ACM international conference on Information and knowledge management. pp. 662–671 (2006)
16. Settles, B.: Active learning literature survey (2009)
17. Settles, B., Craven, M.: An analysis of active learning strategies for sequence labeling tasks. In: proceedings of the 2008 conference on empirical methods in natural language processing. pp. 1070–1079 (2008)
18. Sharma, T., Rattan, D.: Malicious application detection in android — a systematic literature review. Computer Science Review **40**, 100373 (2021)
19. Xu, K., Li, Y., Deng, R., Chen, K., Xu, J.: Droidevolver: Self-evolving android malware detection system. In: 2019 IEEE European Symposium on Security and Privacy (EuroS P). pp. 47–62 (2019). https://doi.org/10.1109/EuroSP.2019.00014
20. Zhang, X., Zhang, Y., Zhong, M., Ding, D., Cao, Y., Zhang, Y., Zhang, M., Yang, M.: Enhancing state-of-the-art classifiers with api semantics to detect evolved android malware. In: Proceedings of the 2020 ACM SIGSAC conference on computer and communications security. pp. 757–770 (2020)

Appendix 11

Publication XI
A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Hybrid feature selection models for machine learning based botnet detection in iot networks. In *2019 International Conference on Cyberworlds (CW)*, pages 324–327, 2019

2019 International Conference on Cyberworlds (CW)

Hybrid Feature Selection Models for Machine Learning Based Botnet Detection in IoT Networks

Alejandro Guerra-Manzanares, Hayretdin Bahsi, Sven Nõmm
Department of Software Science
Tallinn University of Technology, Estonia
e-mail: {alejandro.guerra, hayretdin.bahsi, sven.nomm}@taltech.ee

***Abstract*—Timely detection of intrusions is essential in IoT networks, considering the massive attacks launched by the huge-sized botnets which are composed of insecure devices. Machine learning methods have demonstrated promising results for the detection of such attacks. However, the effectiveness of such methods may greatly benefit from the reduction of feature set size as this may prevent the impeding impact of unnecessary features and minimize the computational resources required for intrusion detection in such networks having several limitations. This paper elaborates on feature selection methods applied to machine learning models which are induced for botnet detection in IoT networks. A particular attention is devoted to the use of wrapper methods and their combination with filter methods. While filter-based feature selection methods provide a computationally light approach to select the most informative features, it is shown that their utilization in combination with wrapper methods boosts up the detection accuracy.**

***Keywords*-Internet of Things, botnet, feature selection, machine learning, wrapper method.**

I. Introduction

The rise of the Internet of Things (IoT) is a reality. There are now over 7 billion IoT devices which may grow up to over 20 billion by 2025 [1]. Their ubiquity combined with the lack of security measures (e.g.: default passwords) have promoted them as an enticing target for cyberattackers, enabling them to be compromised and converted into members of the botnet under the attacker's control, using them as a powerful platform to amplify their attacks [2]. Machine learning (ML) has become a handy and promising approach for the detection of such threats [3]. However, high dimensionality data can yield performance and explainability issues in ML models. Feature selection constitutes an important step in the ML workflow to reduce data dimensionality. Filter feature selection methods are usually the preferred techniques to tackle dimensionality reduction. However, these methods only focus on individual scores of the features without considering the relationship between them. In this sense, wrapper techniques or their combination with filter ones (hybrid methods) may constitute a significant solution.

In this paper, we evaluated the impact of wrapper and hybrid feature selection techniques on the detection accuracy of ML models which are developed for the identification of IoT botnets. Aiming to achieve optimal feature sets that could boost classification while reducing data dimensionality and computational needs. Some studies addressed the application of filter methods to IoT botnet data sets [4] [5]. However, to the best of our knowledge, the elaboration on feature selection methods has not been performed profoundly. This paper fills this research gap by comparing detection performances of filter, wrapper and hybrid methods in this particular problem domain. Section II provides the literature review. Section III explains the methodology while Section IV tackles the main results and discussion. Lastly, Section V concludes the study.

II. Literature Review

Network traffic behavior has been used in combination with ML algorithms for general botnet detection. In [6], filter methods were used to select the best features to build a decision tree model. Flow features were able to discriminate 90.4% malicious traffic in [7]. In [8] eight ML models were assessed with network data. Artificial Fish Swarm algorithm was used in conjunction with Support Vector Machines in [9]. Recurrent neural networks [10] and PSO-based K-means clustering algorithm [11] have also been used to tackle general botnet detection. Fewer approaches have dealt specifically with IoT botnet detection. Logistic Regression algorithm was used in [12] to detect compromised devices. In [13], deep auto-encoders were trained to detect botnet attacks while in [4] the same data set was used focusing on filter methods for feature selection. Five ML algorithms were evaluated in [5] as DoS attack detectors. In [14] Random Neural Networks were trained to detect network attacks against IoT devices. Principal Component Analysis was used in [15] as dimensionality reduction method to induce classification models. As can be noticed, few studies considered feature selection issues on their detection models and none performed any feature selection approach distinct from filter methods.

III. Method

In this study, we followed the ML workflow for a multi-class classification problem focusing on feature subset selection. The steps performed are explained in the following paragraphs.

A. Data set

The data set used in this study has already been used in other studies [4] [13] and it contains both normal and malicious IoT traffic, being preferred for our purpose to other realistic IoT data sets [16]. It is composed of statistics of network traffic collected in a synthetic IoT environment where normal

978-1-7281-2297-7/19/$31.00 ©2019 IEEE
DOI 10.1109/CW.2019.00059

and botnet traffic (Mirai, BashLite) were simulated for nine devices [13]. 115 numeric features are defined for each data point, reflecting aggregated statistics of network raw streams in five time windows (100 ms, 500ms, 1.5s, 10s and 1min, coded as L5, L3, L1, L0.1 and L0.01 respectively) within five major categories, host-IP (traffic originated from specific IP address, coded as H), host-MAC&IP (traffic originated from the same MAC and IP, coded as MI), channel (traffic between specific hosts, coded as HH), socket (traffic between specific hosts, including ports, coded as HpHp) and network-jitter (time interval between packets in channel communication, coded as HH_jit). For each major category, packet count, mean and variance packet size are computed. For socket and channel category, additional statistics are provided such as covariance, correlation coefficient of packet size, radius and magnitude. Distribution of labels in the whole data set is 8% normal traffic, 45% Bashlite and 47% Mirai. As the initial dataset was composed of 9 unbalanced sub-datasets, to create a single, mixed and representative data set, instances from sub-datasets were sampled randomly so that each class is represented equally in the final data set. Features were scaled using inter-quartile range approach, a robust technique to normalize data that contains outlier values (which may negatively affect mean and variance statistics). Stratified 3-fold was applied to split the data set into three equally class balanced folds. Two folds were used in the feature selection steps while the third fold was used in the final stage as a test set.

B. Feature Selection

Feature selection was performed using filter, wrapper and hybrid approaches. They are described as follows:

- Filter methods: a statistical criterion is used to assess the discriminatory power of each feature. The output value is used to select most suitable features. Two filter methods were used in this research, described briefly as follows:

  - Fisher's score: a criterion designed for numeric features that allows to rank them. It measures the ratio of the average inter-class separation to the average intra-class separation.
  - Pearson's correlation coefficient (ρ): a criterion that measures the linear relationship between two variables, ranging from [-1,1].

- Wrapper methods: a heuristic procedure where the performance of the machine learning is used to determine the most relevant feature sets selected by a specific feature search algorithm. Two wrapper methods were used: Sequential Forward Feature Selection (SFFS) and Sequential Backward Feature Elimination (SBFE). Both algorithms attempt to refine a current set of features by adding/removing features iteratively, respectively, based on a classifier's performance. At this step, classifier's performance was evaluated using cross-validated macro-averaged F_1 score. F_1 score is the harmonic mean of precision and recall metrics, ranging $[0, 1]$. The greater the score, the better the model. To select the best minimal subset, a limitation of maximum 5 features on selected subsets was imposed to the wrapper methods evaluated.

- Hybrid approach: a two-step feature selection procedure where the output of a filter method is used as an input for a wrapper method to boost classifier model performance. Both filter methods were combined with both wrapper methods and evaluated with the selected classifier models. This two-step approach is summarized in two sequential steps: 1) Filter step: select the candidate subset according to the filter model (S_F), 2) Wrapper step: evaluate S_F with the classifier model using the macro-averaged F_1 score as a heuristic.

C. Classification model training

In this research, two widely used multi-class classification algorithms were evaluated: k-Nearest Neighbors (k-NN) and Random Forest (RF). As this research focused on feature selection, the attention was centered on these steps, thus providing results on just two traditional ML algorithms make the results easier to analyze and present. In this regard, default parameters of *scikit_learn* library were used: euclidean distance and $k = 5$ for k-NN and $n = 50$ for RF algorithm (maximum depth was limited to 5 to reduce the risk of overfitting).

D. Model validation

Classifiers were validated at two stages using different performance metrics. Two splits of the data set were used in the feature selection part while the third was used to perform models validation as a test data set. These stages are summarized as follows: 1) Feature Selection: wrapper methods use the macro-averaged F_1 score of a classifier to find an optimal subset of features, 2) Models validation: models were built based on feature selection methods output and tested against unseen data. In this step, classification accuracy is reported, which is defined as the ratio of correct predictions among all predictions. In the feature selection step, maximizing F_1 helps to minimize the number of incorrectly classified malicious instances while in the validation step, accuracy is preferred as a more comprehensive metric of the overall performance.

IV. Results & Discussion

A. Feature Selection Results

1) Pearson's ρ: Pairwise linear correlations were computed for all features. To avoid redundancy, all high correlated features were dropped, so that features with $|\rho| \in [0, 0.80]$ were selected. Table I shows the remaining 18 features. 13 out of 18 features belong to the shortest time frame (L5, 100 ms), indicating that the features regarding larger time windows have higher correlations, thus may not add more value to the classification. Only 3 features are host-centric (i.e., Host-MAC&IP) whereas the others are derived from host-to-host communication (i.e., channel, channel jitter and socket categories). This suggests that the statistics obtained from network traffic between host pairs show more uncorrelated behavior. On the other side, most of these features are related to channel or channel jitter categories rather than socket category. Although socket information provides more scrutiny on host-to-host communication by additionally considering port numbers, channel category, which is only based on IP numbers, is enough to generate more discriminatory features.

TABLE I
FEATURES SELECTED USING PEARSON'S ρ

Feature name	Time frame	Major category
MI_dir_L5_weight	L5	Source MAC&IP
MI_dir_L5_mean		
MI_dir_L5_variance		
HH_L5_weight	L5	Channel
HH_L5_std		
HH_L5_radius		
HH_L5_covariance		
HH_L5_pcc		
HH_L0.1_covariance	L0.1	
HH_L0.1_pcc		
HH_L0.01_covariance	L0.01	
HH_L0.01_pcc		
HH_jit_L5_mean	L5	Channel jitter
HH_jit_L5_variance		
HH_jit_L1_variance	L1	
HpHp_L5_weight	L5	Socket
HpHp_L5_radius		
HpHp_L5_covariance		

2) Fisher's score: The Fisher's score (F) of each feature was computed. Best 20 features were selected, as shown in Table II. In contrast to the results of Pearson's method, the best features selected by Fisher's score are host-centric (e.g., Host-IP and Host-MAC&IP categories) and related to weight (packet count). Considering the attacks launched from bots generate a high amount of traffic with varying frequencies (i.e., DDoS, network scans, spam campaigns), it could be perceivable that host-centric statistics would reflect the distinctions between different class behaviors. On the other side, Fisher's score evaluates each feature individually, meaning that it may similarly prioritise features which are correlated to each other. Although the best 4 features belong to longer time intervals (1 minute or 10 seconds), the selected 20 features cover all time intervals, unlike the ones selected by Pearson's ρ.

TABLE II
TOP 20 FEATURES ACCORDING TO FISHER'S SCORE

Rank	Feature name	F	Rank	Feature name	F
1	MI_dir_L0.01_weight	1.54	11	MI_dir_L0.01_mean	0.69
2	H_L0.01_weight	1.54	12	H_L0.01_mean	0.69
3	MI_dir_L0.1_weight	1.23	13	MI_dir_L0.1_mean	0.65
4	H_L0.1_weight	1.23	14	H_L0.1_mean	0.65
5	MI_dir_L1_weight	0.90	15	MI_dir_L1_variance	0.62
6	H_L1_weight	0.90	16	H_L1_variance	0.62
7	MI_dir_L5_weight	0.84	17	MI_dir_L1_mean	0.58
8	H_L5_weight	0.84	18	H_L1_mean	0.58
9	MI_dir_L3_weight	0.84	19	MI_dir_L3_mean	0.53
10	H_L3_weight	0.84	20	H_L3_mean	0.53

3) Filter feature selection models: Classifier models using k-NN and RF were built using subsets chosen by the filter methods. Cross-validated macro-averaged F_1 scores obtained using all features, 4, 10 and 20 best Fisher's score's features and 18 Pearson's features are shown in Table III. The features selected by Fisher's score provided better performance than the ones selected by Pearson's ρ. The use of feature selection in k-NN avoids the curse of dimensionality, providing great performance with a small number of features. 4 features achieved 99.01% detection rate while using all features the performance is reduced, 94.52%. On the other side, RF models demonstrated contrary results as using all features achieved the highest detection rate, 99,96%, and 4 features 97.66%. 20 features with highest Fisher's Score yielded 99.93%, close to the results of all features. It could be argued that as the depth of the trees was limited to avoid overfitting, the performance in the experiments with a smaller amount of features was not optimal. Models using Pearson's ρ features had a low performance with k-NN (91.15%) when compared to the same feature set with RF (98.74%).

TABLE III
CROSS-VALIDATION F_1 SCORE RESULTS ON WHOLE AND FILTERED DATA

Model	Number of features				
	115	4 FS	10 FS	20 FS	18 P
k-NN	0.9452	0.9901	0.9909	0.9940	0.9115
RF	0.9996	0.9766	0.9864	0.9993	0.9874

4) Wrapper models: SFFS and SBFE were used to select best feature subsets using k-NN and RF as classifiers. In order to have a reduced subset of features, an extra limitation was added by establishing the maximum number of features on the selected subset to 5. Results were cross-validated macro-averaged F_1 score, as shown in Table IV. The optimal subset identified by the wrapper method is related to the utilised classifier algorithm as each wrapper-classifier combination results in a different feature set. Both classifiers and wrapper methods produced high performance metrics, F_1 scores over 99.80% in all cases. RF achieved slightly better results than k-NN in each wrapper variation. The optimal subset of features found using SFFS did not reach the maximum set size (5), obtaining high-performance levels with fewer features. Furthermore, in our experiments, we identified that FFS is faster than SBFE taking, on average, 100 times less time to produce an optimal subset. It can be suggested that SFFS is a better choice than SBFE as it has slightly better detection rates, is faster and has smaller optimal feature set sizes.

TABLE IV
WRAPPER METHOD FEATURE SELECTION RESULTS

Wrapper	Classifier	F_1 score	Best subset
SFFS	K-NN, $k = 5$	0.9987	HH_jit_L3_mean HH_L0.01_magnitude HH_L0.01_covariance
	Random Forest	0.9999	H_L0.1_weight H_L0.01_mean HH_jit_L3_mean HpHp_L0.1_magnitude
SBFE	K-NN, $k = 5$	0.9981	HH_L0.01_magnitude HH_L0.01_covariance HH_jit_L0.01_mean HpHp_L0.1_weight HpHp_L0.01_mean
	Random Forest	0.9998	MI_dir_L0.01_variance H_L0.1_mean H_L0.01_weight HH_L0.1_covariance HpHp_L3_magnitude

5) Hybrid models: In this stage, SFFS and SBFE were combined with the outcomes of both filter methods: 20 best Fisher's score features (F) and 18 Pearson's features (ρ). Table V shows the results when hybrid models are built and validated using cross-validation. The performance was over 99% in almost all cases. Best performances were obtained using Fisher's score with RF, where both wrapper methods

achieved the same detection rate, 99.97%. The main advantage produced by the hybrid methods is the reduction of the time needed for the wrapper to produce an optimal subset. Wrapper methods use all features as an input, which can be troublesome and time-consuming with large sets of features. Providing a reduced set of features produces faster results. Although in some cases there is a slight reduction of F_1 score in the hybrid method compared with the wrapper methods, there is a great time reduction on hybrid methods, producing optimal subsets 10 times faster using hybrid SFFS than wrapper SFFS and 100 times faster using hybrid SBFE than wrapper SBFE.

TABLE V
CROSS-VALIDATION F_1 SCORES ON HYBRID FEATURE SELECTION MODELS

Wrapper	Filter	k-NN	Random Forest
SFFS	F	0.9993	0.9997
	ρ	0.9916	0.9965
SBFE	F	0.9989	0.9997
	ρ	0.9871	0.9967

B. Test fold validation

All combinations of feature selection methods were validated on a test set that simulates the prediction of unseen data. Test accuracy values are shown in Table VI. The first two columns specify the filter and wrapper method used (-, if none). In the case of the wrapper and hybrid methods, the selected subset was tested using the two classifier algorithms, not just the specific algorithm used in the feature selection method step to obtain the corresponding subset (cross-classifier tests). High accuracy values (i.e., over 98.00% in most cases) were obtained using filter methods and wrapper methods alone, supporting the goodness of feature selection methods to achieve high performance metrics with reduced data input. However, the highest accuracy values were obtained using the combination of filter and wrapper methods (hybrid techniques). More specifically, the use of Fisher's score with any of the wrapper methods and classifiers, provided the best performances in almost all cases, over 99.90% accuracy even in cross-classifier tests. In this regard, best results are achieved using the RF as classifier except the cases in which SFFS is utilised in combination with k-NN as a wrapper model.

TABLE VI
TEST FOLD ACCURACY COMPARISON OF ALL MODELS

Filter method	Wrapper method	k-NN	Random Forest
-	-	0.9536	0.9985
FS	-	0.9968	0.9990
ρ	-	0.9224	0.9852
-	SFFS k-NN	0.9982	0.9608
-	SFFS RF	0.9986	0.9988
-	SBFE k-NN	0.9984	0.9788
-	SBFE RF	0.9974	0.9992
FS	SFFS k-NN	0.9994	0.9992
	SFFS RF	0.9938	0.9994
	SBFE k-NN	0.9990	0.9992
	SBFE RF	0.9992	0.9992
ρ	SFFS k-NN	0.9912	0.8475
	SFFS RF	0.9622	0.9972
	SBFE k-NN	0.9906	0.9958
	SBFE RF	0.9013	0.9970

V. CONCLUSIONS

The present paper has demonstrated the applicability and importance of the hybrid feature selection technique to the problem of ML based IoT botnet detection. Application of the hybrid feature selection technique may be seen as the trade-off between the simplicity of filter models based feature selection and the more computationally demanding wrapper techniques. It was demonstrated that hybrid feature selection allows reducing the computational load of the wrapper techniques without any significant loss in detection rates of the machine learning classifiers. More specifically, the combination of Fisher's score with wrapper methods provided consistently higher accuracy rates in each of the classification models.

REFERENCES

[1] IoT-Analytics, "State of the IoT 2018: Number of IoT devices now at 7B Market accelerating." [Online]. Available: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/

[2] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "Ddos in the iot: Mirai and other botnets," *Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[3] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2015.

[4] H. Bahşi, S. Nõmm, and F. B. La Torre, "Dimensionality reduction for machine learning based iot botnet detection," in *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*. IEEE, 2018, pp. 1857–1862.

[5] R. Doshi, N. Apthorpe, and N. Feamster, "Machine learning ddos detection for consumer internet of things devices," in *Security and Privacy Workshops (SPW)*. IEEE, 2018, pp. 29–35.

[6] D. Zhao, I. Traore, B. Sayed, W. Lu, S. Saad, A. Ghorbani, and D. Garant, "Botnet detection based on traffic behavior analysis and flow intervals," *Computers & Security*, vol. 39, pp. 2–16, 2013.

[7] L. Bilge, D. Balzarotti, W. Robertson, E. Kirda, and C. Kruegel, "Disclosure: detecting botnet command and control servers through large-scale netflow analysis," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 129–138.

[8] M. Stevanovic and J. M. Pedersen, "An efficient flow-based botnet detection using supervised machine learning," in *International conference on computing, networking and communications*. IEEE, 2014.

[9] K.-C. Lin, S.-Y. Chen, and J. C. Hung, "Botnet detection using support vector machines with artificial fish swarm algorithm," *Journal of Applied Mathematics*, vol. 2014, 2014.

[10] P. Torres, C. Catania, S. Garcia, and C. G. Garino, "An analysis of recurrent neural networks for botnet detection behavior," in *2016 IEEE biennial congress of Argentina (ARGENCON)*. IEEE, 2016, pp. 1–6.

[11] S.-H. Li, Y.-C. Kao, Z.-C. Zhang, Y.-P. Chuang, and D. C. Yen, "A network behavior-based botnet detection mechanism using pso and k-means," *ACM Transactions on Management Information Systems (TMIS)*, vol. 6, no. 1, p. 3, 2015.

[12] A. O. Prokofiev, Y. S. Smirnova, and V. A. Surov, "A method to detect internet of things botnets," in *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. IEEE, 2018, pp. 105–108.

[13] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, A. Shabtai, D. Breitenbacher, and Y. Elovici, "N-baiot - network-based detection of iot botnet attacks using deep autoencoders," *IEEE Pervasive Computing*, vol. 17, no. 3, pp. 12–22, 2018.

[14] O. Brun, Y. Yin, and E. Gelenbe, "Deep learning with dense random neural network for detecting attacks against iot-connected home environments," *Procedia computer science*, vol. 134, pp. 458–463, 2018.

[15] S. Zhao, W. Li, T. Zia, and A. Y. Zomaya, "A dimension reduction model and classifier for anomaly-based intrusion detection in internet of things," in *3rd Intl Conf on Big Data Intelligence, Computing, Cyber Science and Technology Congress*. IEEE, 2017, pp. 836–843.

[16] N. Koroniotis, N. Moustafa, E. Sitnikova, and B. Turnbull, "Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset," *Future Generation Computer Systems*, vol. 100, pp. 779–796, 2019.

Appendix 12

Publication XII
A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. Towards the integration of a post-hoc interpretation step into the machine learning workflow for iot botnet detection. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 1162–1169, 2019

2019 18th IEEE International Conference on Machine Learning and Applications (ICMLA)

Towards the Integration of a Post-hoc Interpretation Step into the Machine Learning Workflow for IoT Botnet Detection

Alejandro Guerra-Manzanares, Sven Nõmm and Hayretdin Bahşi
Department of Software Science, Faculty of Information Technology
Tallinn University of Technology, Akadeemia tee 15 a, 12618,
Tallinn, Estonia
E-mail: {alejandro.guerra, sven.nomm, hayretdin.bahsi}@taltech.ee

***Abstract*—The analysis of the interplay between the feature selection and the post-hoc local interpretation steps in a machine learning workflow followed for IoT botnet detection constitutes the research scope of the present paper. While the application of machine learning-based techniques has become a trend in cyber security, the main focus has been almost on detection accuracy. However, providing the relevant explanation for a detection decision is a vital requirement in a tiered incident handling processes of the contemporary security operations centers. Moreover, the design of intrusion detection systems in IoT networks has to take the limitations of the computational resources into consideration. Therefore, resource limitations in addition to human element of incident handling necessitate considering feature selection and interpretability at the same time in machine learning workflows. In this paper, first, we analyzed the selection of features and its implication on the data accuracy. Second, we investigated the impact of feature selection on the explanations generated at the post-hoc interpretation phase. We utilized a filter method, Fisher's Score and Local Interpretable Model-Agnostic Explanation (LIME) at feature selection and post-hoc interpretation phases, respectively. To evaluate the quality of explanations, we proposed a metric that reflects the need of the security analysts. It is demonstrated that the application of both steps for the particular case of IoT botnet detection may result in highly accurate and interpretable learning models induced by fewer features. Our metric enables us to evaluate the detection accuracy and interpretability in an integrated way.**

***Index Terms*—Botnet detection, machine learning, interpretation.**

I. Introduction

This paper aims to provide interpretable results for machine learning-based IoT botnet detection system while minimizing the feature set. IoT technology has been an important enabler in many sectors such as energy, manufacturing, transportation or health. However, physical-, network-, and application-layer attacks may cause important consequences ranging from privacy violations or business interruptions to physical damage in these systems [1]. Besides these problems, the botnets composed of compromised IoT devices constitute a significant threat to all Internet-faced systems. For instance, in 2016, some companies such as a hosting company, OVH [2] and an internet performance management company, Dyn DNS [3], suffered from massive denial of service attacks originated from IoT bots. Considering the high adaptation rate of IoT technology into real-world applications, these attacks could be assumed as initial warnings for the more detrimental attacks in the future. The research community has addressed the intrusion detection in these environments [4]. Identifying the new attack types remains as a significant difficulty, but machine learning methods have provided solutions for this problem [5].

Although there exists a huge amount of research with convincing results for the adaptation of machine learning to the intrusion detection field, the optimization of the detection accuracy has been the only focus in those studies. However, the detection result is just only a starting phase of an incident handling process that mostly takes place in a layered tier structure of the security operations center [6]. In an ideal case, a full-automatized protection system itself should identify the intrusion, decide and take the necessary actions. However, human-in-the-loop character of these systems has not reduced, inversely, increased in time due to the complexity of the cyber threats. The responsibility of experts is so vital in terms of reducing the false positives, assigning priority levels to the findings and conducting in-depth forensic analysis during the incident handling period. Therefore, machine learning method is highly required to provide the details of explaining the detection decision to the experts, which means interpretability arises as a significant performance metric besides the accuracy rate of the detection.

A model in which the expert can easily understand the reason behind the decision is considered as an interpretable model (a.k.a. comprehensible, understandable or explainable [7]). Reducing the model size via dimensionality reduction is one of the approaches for making models more interpretable [8] despite the fact that it does not guarantee the acceptability in all cases (i.e., the experts may not trust over-simplistic models in some situations) [9]. Interpretability is an ill-defined concept as some scholars relate it to a more general notion such as trust of users to learning models, some of them find it as an instrument for identifying the casual structure or others may simply assume that it helps to gather more useful data from the model [10].

Interpretability methods can be divided into two, global or local. Global interpretability aims to provide understanding of the whole logic of a model and all of its possible outcomes while local interpretability deduces the reasoning behind an individual prediction. In practice, the global interpretability of a model is hardly achievable while local interpretability being more feasible.

Application of the classical machine learning techniques

978-1-7281-4550-1/19/$31.00 ©2019 IEEE
DOI 10.1109/ICMLA.2019.00193

usually presumes that the feature selection process was conducted[1]. Model-agnostic and post-hoc local interpretation methods are applied to the outputs of learning models. In this sense, feature selection is a prior step and local interpretation is the following step for the creation and validation of the learning models in classical algorithms. Usually, local interpretation for each particular instance is presented by the set of inequalities (i.e., assuming that the features are numeric), whereas the number of inequalities is greater or equal to the dimensionality of the feature set. Therefore, although they take place in different steps, feature selection and local interpretation may have an interplay, in the whole machine learning work-flow, that may have an impact on the quality of the interpretation. Besides this, it is obvious that the feature selection has also an impact on the accuracy of the model. Intrusion detection designers would prefer to understand the implication of the feature selection not just only to the accuracy but to the quality of the interpretation as well.

In this study, we analyzed the impact of feature selection on the detection accuracy and the interpretation quality. In the first part, we analyzed the impact of hyper-parameters and feature selection on data accuracy. In the second part, we analyzed the impact of feature selection on the interpretation results. Here, we introduced a quality metric for the interpretation results from the cyber security analyst perspective. This metric, which is based on entropy notion, assumes that an ideal explanation should bring explanation for just only one category as others could create confusion for the analyst. As our focus is not the optimization of the local interpretation step but just to understand the interplay between feature selection and post-hoc local interpretation, we used well-known Local Interpretable Model-agnostic Explanation (LIME) as a local interpretation method [11] although there are other methods in the literature. The decision quality of the interpretation method and comparing it with other similar methods are also out of our scope.

Recently deep-learning-based techniques have gained a lot of popularity. Unlike the classical machine learning algorithms, the deep-learning methods do not require separate feature selection procedure[2]. However, as IoT environments are limited with various computational resource restrictions, classical learning models induced by fewer features can be considered more suitable than deep-learning methods which are computationally intensive. The very high accuracy values that we obtained by classical methods in this problem domain reinforce our method selection decision.

The studies dealing with the application of machine learning to the cyber security problems, in general, and to the intrusion detection, in particular, give the whole focus to the detection accuracy, and very little attention is paid to the interpretability. This paper addresses such an important research gap. Our work is unique as it provides an interplay analysis of feature selection and interpretability steps within the context of the IoT botnet detection problem.

The content of our paper is presented as follows: Section II gives background information about the addressed topic and summarizes the literature. In Section III, we described the data utilized in this study. Section IV explains the method of our research and the obtained results. Our work is concluded in Section V.

II. Background Information and Literature Review

Artificial Intelligence has become a widely adopted solution to deal with some complex tasks such as prediction in a great variety of fields ranging from biology to cybersecurity. The inherent complexity of most machine learning models makes them powerful but lacking transparency, posing as black-boxes, where the explanation each decision remains hidden and unknown to the end-user. Thus, becoming one of the main obstacles to the spreading of AI to fields where the explanation behind each decision is important and needed to trust and justify the AI-system predictions, for instance in disease diagnosis or compliance with General Data Protection Regulation (GDPR).

In recent years, explainable AI (XAI) has emerged as a new research field aiming to create more human-interpretable models, that will empower transparency and trust in machine learning outcomes, whilst preserving their high-performance capabilities [12]. The main issue for XAI models is dealing properly with the interpretability and accuracy tradeoff. One of the successful approaches that allows keeping high-performance metrics while providing interpretability to complex machine learning models is post-hoc methods, which provide explanations without disturbing or having any knowledge about the inner works of the model they are explaining [12].

Local interpretation methods have focused the attention especially to explain the complex and opaque Deep Neural Networks, methods which usually claim to be model-agnostic [12]. In this regard, [11] proposed the Local Interpretable Model-Agnostic Explanation (LIME) method which explains individual instances of non-linear models by sampling perturbed instances around the individual decision, weighting them according to their proximity, getting original model's output for the perturbed instances and learning a linear model in the neighbourhood of the explained instance. From the same authors and more recently, in [13] they proposed Anchors, which extend LIME's model-agnostic explanations on the basis of if-then rules, claiming to provide more coverage, interpretability, and enhanced generalization. Local Rule-based Explanations (LORE), proposed in [14], builds a decision tree model based on a set of neighbor instances of a concrete decision, using a genetic algorithm and given an original black-box model. Explanation of individual decision is extracted from the learned decision tree. In [15], local explanations are based on the local gradients which identify what directions an instance has to be moved in order to change its predicted label, thus

[1]While some techniques do not suffer from the curse of dimensionality, feature selection is still necessary to provide stable classifiers.

[2]It may be seen that feature selection becomes an implicit integral part of the deep learning process

indicating the most influential directions to the prediction. Lastly, [16] applied influence functions that measure the effect of local changes in instances to understand individual predictions.

Cybersecurity is a potential field of application where XAI models are needed [12]. Regarding it, network security is one of the emerging issues where interpretability may help to explain and improve detection mechanisms such as intrusion detection systems. In this regard, the framework for explainable Deep Neural Networks (DNN) based anomaly detection implemented in [17] relied on input feature relevance scores on network-based anomaly detection to explain individual decisions and enhance human trust on machine learning models in the context of critical and industrial systems monitoring. Input relevance scores measured quantitatively the influence of certain input features on the detected anomaly by the DNN model. In the same context, but as a different approach, Situ [18] proposed the use of p-value anomaly scoring to explain anomalies detected on network data streams. P-value anomaly scoring was calculated on a set of statistics of interest and used to discriminate and explain network suspicious behavior. Adversarial learning samples, mainly used to deceive ML classifier models, are adopted in [19] to explain Deep Learning models used as classifiers in Intrusion Detection Systems (DNN-IDS). In this regard, a model-agnostic adversarial machine learning method is implemented to explain misclassified samples by feeding the classifier with misclassified samples, making minimum amount of modifications on them until they are correctly classified. Comparison of modified samples correctly classified and original samples is used to find out the most relevant features that produced the misclassification, thus explaining the classifier's output.

III. Data set description

The dataset is composed of various statistics obtained from the network traffic of 9 IoT devices such as a security camera, webcam, baby monitor, thermostat, and door-bell [20]. Each record contains 115 numeric features and is labeled as benign (normal) or malicious (attack). The malicious traffic covers different attacks (i.e., denial of service, spam or scan) which are conducted by the devices compromised with Mirai or Bashlite (Gafgit) malware. In a typical botnet life-cycle, there exist four phases, formation, command and control (C&C), attack and post-attack [21]. This dataset does not include malicious activities that are related to the first two phases which correspond to exploitation of the device and creation of a remote control channel but covers the post-exploitation activities (attack and post-attack phases).

We analyzed the features within five categories, host-IP, host-MAC&IP, channel, network jitter and socket as shown in Table I. Host-IP category tracks the network traffic of each host regardless of the other communication entity and provides the statistics such as packet counts, mean and variance of packet sizes. Host-MAC&IP category is very similar to the previous one, the only difference is that it dissects each host by its MAC and IP addresses in order to eliminate the artifacts of possible IP spoofing attempts. Channel category captures the sames statistics produced by the source and destination host pairs whereas socket category also includes the source and destination ports in addition to host information. The statistics of the last two categories are extended by magnitude, radius, covariance and correlation coefficient of packet sizes. We classified the network-jitter of the channel type communication (i.e., the time intervals between the packet arrivals) into a separate category. All these statistics are obtained from the most recent five different time windows (100ms, 500ms, 1.5 sec, 10 sec and 1 min). In order to improve the readability, we represent a feature as "Feature Category Type-Time Window-Statistic Type". For instance, "Host_IP-100ms- Pkt Count" means the packet count of host-IP category obtained at the most recent 100ms interval.

The source dataset has 502,605 normal, 2,835,317 Bashlite and 2,935,131 Mirai records, meaning that the label distributions are 8%, 45%, and 47%, respectively. In this study, we covered the three-class classification problem. We utilized the accuracy as a detection metric for the simplicity as we focus on the interaction between feature selection and interpretation steps, and such an analysis can be conducted by other metrics deemed to be useful.

TABLE I
Feature Categories

Feature Categories	Features
Host-IP	Packet count, mean and variance (outbound)
Host-MAC&IP	Packet count, mean and variance (outbound)
Channel	Packet count, mean and variance (outbound) Magnitude, Radius, Covariance, Correlation Coef. (inbound and outbound)
Network Jitter	Count, mean and variance of packet jitter in channel
Socket	Packet count, mean and variance (outbound) Magnitude, Radius, Covariance Correlation Coefficient (inbound and outbound)

IV. Method & Results

The feature selection step is an integral part of any ML workflow where a classification algorithm is used. According to [22] it may be either a stand-alone step or integrated as a part of a wrapper or an embedded technique. Within the frameworks of the present research, only the case when feature selection is a stand-alone step is considered. Namely Fisher's score (given in Equation 1) based filter model is applied to provide initial grading of the features with respect to their discriminating power. Such simplification is possible due to the numeric nature of the features.

A typical work-flow for the application of supervised learning method is depicted in Figure 1. According to [22] three different approaches may be used for feature selection: filter models, wrapper models and embedded models.

Fisher's score is closely related to the information gain and usually defined as given in Equation 1.

$$F_i = \frac{\sum_{k=1}^{K} p_k(\mu_k - \mu)^2}{\sum_{k=1}^{K} p_k \sigma_k^2} \quad (1)$$

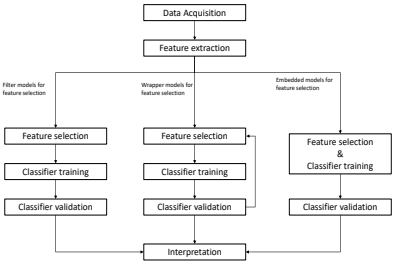


Fig. 1. Usual machine learning work-flow for supervised learning.

where K is the number of classes, p_k proportion of the observation points belonging to the class k, μ_k is the mean value of the class k along the feature i, μ is the overall mean and σ_k is the standard deviation of the class k along the feature i [22]. The higher value of (6) signals greater discriminating power of the feature and vice verse. Fisher's score is computed for each of the 115 features. The features are ordered according to their Fisher's scores as shown in Figure 2. Then feature selection may be performed either with respect to certain threshold or simply by choosing desired number of features based on their Fisher's score values. In Figure 2 one may clearly see that Fisher's score values for some features are negligibly low which is a clear indicator that these features may be omitted.

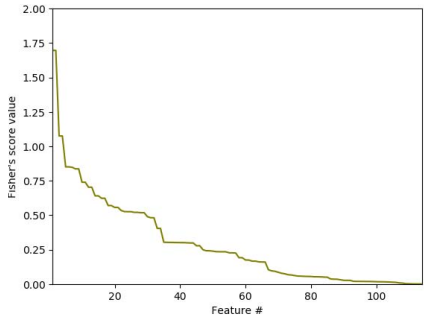


Fig. 2. Values of Fisher's score ordered in descending order.

A. Optimization of Model Hyper-Parameters

Detection of botnet traffic is considered in this subsection as the classification problem where three classes are considered. The first class corresponds to the benign traffic (normal operation of the devices), the second class is Mirai attack and the third class corresponds to the Bashlite attack. Decision trees (DT), k - nearest neighbors (kNN), and logistic regression (LR), are frequently considered as classical classification techniques. This list may be complemented by random forest classifiers (RF) and support vector machines (SVM). Usually these techniques require lesser levels of computational power than deep learning techniques. The application of these methods leads different trade-offs between the hyper-parameters of the algorithms their accuracy. Let us first consider how number of features affects overall accuracy.

For the case of kNN, number of nearest neighbors k and number of features are the hyper-parameters. In Figure 3 accuracy of this method computed during 5-fold cross-validation for different values of hyper-parameters is depicted. The most optimal value for the number of nearest neighbours

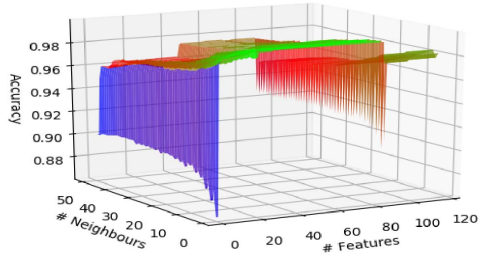


Fig. 3. Trade-offs between accuracy and hyper-parameters in kNN classifiers

is $k = 1$. In the framework of the present research, the depth of the decision trees was not limited in any manner, therefore, the number of the features is the sole hyper-parameter. Dependence between the number of the features and accuracy of the DT classifier is depicted in Figure 5.

For the RF classifier, there are two hyper-parameters number of the features and number of the trees. Like in the case of decision trees depth of the individual trees was not limited in any way. Figure 4 depicts the accuracy as the function of the number of features and number of trees. The accuracy of the classifiers is stable high when number of the features is greater or equal 11 and number of trees is greater or equal than 11.

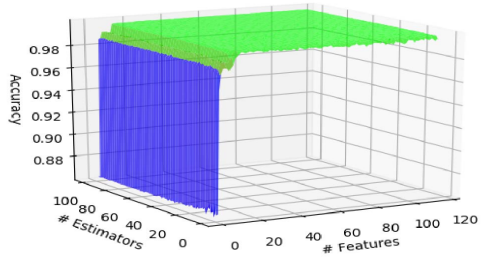


Fig. 4. Trade-offs between accuracy and hyper-parameters in RF classifiers

Figure 5 demonstrates the change in accuracy with the number of the ordered features in learning models induced

by DT, RF and kNN (i.e., note that such values are derived from the learning models with the hyper-parameters providing highest accuracy). Logistic regression and support vector machine classifiers have demonstrated lower accuracy rates in comparison to the other methods and therefore omitted from further studies.

Observing the trade-offs between the accuracy and number of features depicted in Figure 5, one may easily see that it is not necessary to base the classification process on all the available features. A feature set having 10-15 features provide optimal accuracy performance. Results of the present section clearly demonstrate that, in our problem, smaller number of features is enough to reach very high accuracy rates, which eliminates the need to use deep-learning algorithms requiring high computational resources. It is important to note that, in most of the cases, intrusion detection in IoT network should be done with limited resources due to the hardware constraints in IoT devices, justifying the minimization of required resources for inducing learning models.

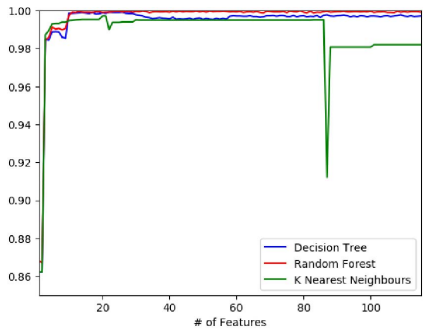


Fig. 5. Comparison of the trade-off between the number of features and accuracy

B. LIME based interpretation

We applied the LIME method [11] to the instances labeled by learning models. In order to compare the local explanations generated for the outputs of these models, we conducted several experiments by using supervised learning methods such as Random Forest, k-NN and Decision Tree with varying sizes of selected features. If not otherwise stated, we chose the hyper-parameters that utilize the best accuracy according to the results of experiments given in Section IV-A. Table II gives the explanations of three test instances belonging to the categories, Benign, Bashlite and Mirai which are correctly classified by the corresponding model (the data points are [4.53 4.53 2.19], [480,00 480.00 219.69] [798.87 798.87 267.78] respectively). In Table II letter 'w' in the feature name column refers to 'weight'. The value of k was chosen as 3 for kNN and the number of estimators was utilized as 10 in Random Forest. The learning models were induced with the best three features selected by the Fisher's Score. In the data-set, the features that belong to Host-IP and Host-MAC&IP categories are very similar. Therefore, local explanations for the same corresponding features (for instance, MI_dir_L1_weight and H_L1_weight) have the same value. It can be observed that all learning methods provide similar intervals for the selected test instances. In case of k-NN, for example, a security analyst can easily deduce from the explanations that if the packet count of a host captured in 1.5 sec and 500 ms intervals are lower than 277.96 and 112.94 respectively, then that host is not compromised by any malware type. Although the acquired explanations are not global, meaning that they do not represent the whole characteristics of the corresponding category, and they are just local explanations of the selected test instances, the results can give intuition to the analyst about the difference between Bashlite and Mirai categories (i.e., Bashlite infection produced more packets then Mirai infection).

TABLE II
EXPLANATIONS FOR SELECTED TEST INSTANCES

Label	**Feature**	**kNN** ($k=3$)	**Rand. For.** ($10Est.$)	**Decision** tree
Benign	MI_dir_L1_w	≤ 277.96	≤ 268.37	≤ 270.17
	H_L1_w	≤ 277.96	≤ 268.37	≤ 270.17
	MI_dir_L3_w	≤ 112.94	≤ 110.04	≤ 110.21
Bashlite	MI_dir_L1_w	≥ 679.91	≥ 680.42	≥ 677.42
	H_L1_w	≥ 679.91	≥ 680.42	≥ 677.42
	MI_dir_L3_w	≥ 246.09	≥ 247.44	≥ 244.75
Mirai	MI_dir_L1_w	> 277.96	> 268.37	> 270.17
		≤ 595.68	≤ 594.29	≤ 593.59
	H_L1_w	> 277.96	> 268.37	> 270.17
		≤ 595.68	≤ 594.29	≤ 593.59
	MI_dir_L3_w	> 193.25	> 191.15	> 194.09
		≤ 246.09	≤ 247.44	≤ 244.75

We applied LIME to the randomly selected 50 test instances from each category. The results of correctly categorised instances are given in Table III (note that f1 refers to the feature, MI_dir_L1_weight and f2 refers to MI_dir_L3_weight). As the Host-IP and Host-MAC&IP categories have the same values for the same statistical feature, we did not include the Host-IP category in the table. Each row gives the inequality set for the explanation and the number of instances explained by such an exactly the same inequality set. The results show that all instances belonging to the benign category are represented by one common inequality set whereas five and eight inequality sets are created for Bashlite and Mirai, respectively. As the identified explanations are local and the models are trained with only three features, we observed the same exact explanations for the instances of different categories as one may expect (namely, explanation overlap). The rows shaded by the same gray tone show the same inequality set that is recognized in different categories. For instance, all benign, seven Bashlite and seven Mirai instances are explained by the same inequality set, $MI\_L1\_weight \leq 277.96$,MI_L3_weight≤ 112.94. Another similar result is obtained for the set, $MI\_L1\_weight > 679.91$, MI_L3_weight> 246.09, which explains 26 Bashlite and 7 Mirai instances.

It is obvious that providing the same explanation to different classes would create big confusion for the security analysts

TABLE III
SAMPLE EXPLANATIONS FOR THE OUTPUT OF K-NN

Class	Inst.	Explanation Rules
Benign	50	f1 $\leq$ 277.96, f2 $\leq$ 112.94
	7	f1 $\leq$ 277.96, f2 $\leq$ 112.94
	4	f1 > 679.91, 193.25 < f2 $\leq$ 246.09
Bashlite	26	f1 > 679.91, f2 > 246.09
	1	f1 $\leq$ 277.96, f2 $\leq$ 193.25
	1	277.96 < f1 $\leq$ 595.68, 112.94$\leq$ f2 $\leq$ 193.25
	4	277.96 < f1 $\leq$ 595.68, 193.25 < f2 $\leq$ 246.09
	12	277.96 < f1 $\leq$ 595.68, 112.94 < f2$\leq$ 193.25
	7	f1 > 679.91, f2 > 246.09
Mirai	12	595.68 < f1 $\leq$ 679.91,193.25 < f2$\leq$ 246.09
	7	f1 $\leq$ 277.96, f2 $\leq$ 112.94
	3	595.68 < f1 $\leq$ 679.91,112.94 < f2 $\leq$ 193.25
	1	f1 $\leq$ 277.96,112.94 < f2 $\leq$ 193.25
	1	277.96 < f1 $\leq$ 595.68, f2 $\leq$ 112.94

even if the accuracy of the model is so high. The hyper-parameter choices, including the number of selected features, applied at each step before the post-hoc interpretation would definitely have an impact on the overlaps of the local explanations. In this work, we focus on the implications of these choices rather than the decision quality of the interpretation algorithm itself. In order to show the explanation overlap generated by the LIME interpretation in a better way, we restructured the sample data given in Table III as shown in Table IV so that each row gives a distinct explanation with the distribution of the instance categories described by that explanation. For instance, the first row, $f1 \leq 277.96, f2 \leq 112.94$, explains 50 benign, 7 Bashlite and 7 Mirai instances, which is not the ideal case for an analyst as the same inequality explains three categories. On the other side, the second row, $f1 > 679.91, 193.25 < f2 \leq 246.09$ just explains 4 Bashlite instances but no benign and Mirai ones, not creating a confusion.

TABLE IV
CLASS DISTRIBUTION FOR EACH EXPLANATION (LEARNING MODEL IS K-NN)

Explanation Rule	Benign	Bashlite	Mirai
f1 $\leq$ 277.96, f2 $\leq$ 112.94	50	7	7
f1 > 679.91, 193.25 < f2 $\leq$ 246.09	0	4	0
f1 > 679.91, f2 > 246.09	0	26	7
f1$\leq$ 277.96, f2 $\leq$ 193.25	0	1	0
277.96 < f1 $\leq$ 595.68, 112.94 $\leq$ f2 $\leq$ 193.25	0	1	12
277.96 < f1 $\leq$ 595.68, 193.25 < f2 $\leq$ 246.09	0	0	4
595.68 < f1 $\leq$ 679.91,193.25 < f2 $\leq$ 246.09	0	0	12
595.68 < f1 $\leq$ 679.91,112.94 < f2 $\leq$ 193.25	0	0	3
f1 $\leq$ 277.96,112.94 < f2$\leq$ 193.25	0	0	1
277.96 < f1 $\leq$ 595.68, f2$\leq$ 112.94	0	0	1

We introduced an interpretability quality metric in Equation 2 which computes the degree of explanation overlap by using the entropy notion. Assume that e_i is the ith explanation in an explanation set E, K is the number of categories and p_k is the ratio of instances labeled by category k to the all instances described by e_i.

$$e_i = \sum_{k=1}^{K} -p_k \cdot \log_2 p_k \tag{2}$$

The value of $p_k \cdot \log_2 p_k$ is considered as zero when p_k is zero. Equation 2 gets the lowest value, zero, when all instances explained with one inequality belongs to the same category (namely, explanation overlap is zero) and provides the highest value in case of instances are equally distributed among the categories. Therefore, the first, third and fifth rows in Table IV have entropy values greater than zero whereas all others are exactly zero.

Let's assume that we have N instances and apply LIME to get an explanation for each instance. The explanation overlap, E, of an entire explanation set having N elements is computed as follows:

$$E = \sum_{i=1}^{N} e_i \tag{3}$$

where e_i is computed by Equation 2.

Figure 6 shows the explanation overlap (EO) of a randomly selected instance set (recall that we selected 50 from each category) and explained by LIME for the learning models created by decision trees, kNN and random forest with varying selected features. The x-axis of the graph gives the number of features ordered according to the Fisher's Score and the y-axis demonstrates the value of explanation overlap obtained by the chosen features (i.e., using Equation 3). The results show that all machine learning methods reach the zero value for EO between 13 and 17 features (i.e., although all of them provide non-zero values for some greater feature numbers), meaning that, at the post-hoc interpretation step, the LIME requires at least such number of features to assign one explanation to just only one category. If the machine learning model utilizes fewer features, the explanations may, in turn, confuse the analysts so that one inequality set may explain more than one category. On the other side, once reaching the zero value, LIME provides more zero-valued results for the interpretation of the decision tree models for the greater number of features as there exist fewer fluctuations in the remaining of the graph for this learning model. However, RF and kNN still give much non-zero EO values after 13-17 features.

Recall, in Figure 5, it is shown that the learning models have already reached the optimal accuracy values around 10-15 features. Therefore, we can deduce that it is possible to have a clear explanation figure and optimal accuracy with 13-17 features in our problem. However, such a number of features can not be comprehensible by the experts as the feature set has so many inequalities. Miller's psychological theory states that humans can handle $7(+/-)2$ abstract entities at the same time [23]. Although there is no clear definition of interpretability (i.e., whether it includes comprehensibility or acceptability), it is obvious that 13-17 features may not be preferable by the experts in spite of the high detection accuracy rates. Such high

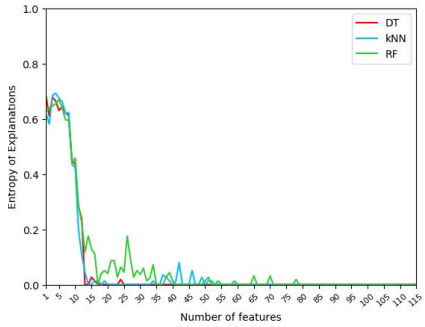


Fig. 6. Equation overlap results by using the features ranked by Fisher's Score

numbered features can not easily explain the decision to the human analyst even in the decision trees which are considered as one of the most inherently interpretable models.

Additionally, the dataset has various features belonging to different time-variants but seem semantically similar, which may be very hard for analysts to dissect the categories. Figure V shows the best 20 features selected by Fisher's score. It can be observed that most of the packet counts in different time intervals (for instance, "MI_dir_L1_weight", "MI_dir_L3_weight" or "MI_dir_L0.1_weight") are determined as the best features which are, in turn, included in the interpretation sets. For an analyst, analyzing all of them may not be much sense if there are no big behavioral deviations in such different intervals. The similarity of data in feature categories, Host-IP and Host-MAC&IP, makes the comprehensibility issue more problematic.

TABLE V
SELECTED FEATURES BY FISHER'S SCORE

Features (in a descending order)
MI_dir_L1_weight
H_L1_weight
MI_dir_L3_weight
H_L3_weight
MI_dir_L0.1_weight
H_L0.1_weight
MI_dir_L5_weight
H_L5_weight
MI_dir_L0.01_weight
H_L0.01_weight
HH_L1_weight
HH_jit_L1_weight
MI_dir_L1_variance
H_L1_variance
HH_L3_weight
HH_jit_L3_weight
MI_dir_L0.01_mean
H_L0.01_mean
MI_dir_L0.1_variance
H_L0.1_variance

Here, it is important to note that the rank of Fisher's score can not be treated as ground truth for the determination of discriminatory power. Wrapper or embedded feature selection methods may yield better accuracy rates with less number of features. One alternative could be also to eliminate the dependent features. Nevertheless, the comprehensive analysis of feature selection methods is beyond the scope of the paper. However, we conducted additional experiment to see the results of another feature set which we selected in the following way: We traversed the list ranked by Fisher Score, but included the 20 features that belong to different feature category (i.e., refer to Table I for these categories) in our final list given in Table VI . We also eliminated the "Host-MAC&IP" category due to its similarity to the "Host-IP" category. This selection means that the final list also includes features with the lower Fisher's Score.

TABLE VI
CUSTOMARY SELECTION OF THE FEATURES

Features (in a descending order)
H_L1_weight
HH_L1_weight
HH_jit_L1_weight
H_L1_variance
H_L0.01_mean
HH_L0.01_std
HpHp_L0.01_mean
HH_L1_mean
HH_jit_L5_mean
HH_jit_L0.01_variance
HpHp_L0.01_std
HpHp_L0.01_magnitude
HH_L0.01_magnitude
HH_L0.01_radius
HH_L0.01_pcc
HpHp_L0.01_radius
HpHp_L5_covariance
HpHp_L5_pcc
HH_L0.1_covariance
HpHp_L0.1_weight

Explanation overlap and accuracy results for the custom feature set are given in Figures 7 and 8. It is interesting to note that the LIME interpretation of all models reached zero with 6 features for the value of equation overlap, and there does not exist any fluctuation in the remaining part of the graph for greater numbers of features. When the results in Figures 8 and 5 are compared, it can be deduced that the optimal detection accuracy is already reached with 5 features, and the overall accuracy figure is similar to the previous case in which the strictly ranked feature set was used. The number of selected features is still within the range of limits stated in Miller's theory. As the features belong to different categories, it can be argued that the security analysts could better perceive the interpretation rules and understand the distinctions between the benign and malware types.

Although we have not thoroughly investigated all feature selection methods, the reduction in the size of the optimal feature set from the accuracy point of view could be attributed to the possible dependencies among features or it can be argued that a filter method which is computationally cheap is not enough. However, the quality metric, explanation overlap, that we proposed in this study, supported the interpretability analysis of the selected features.

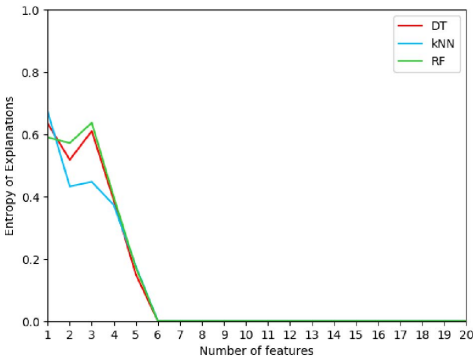


Fig. 7. Using custom features

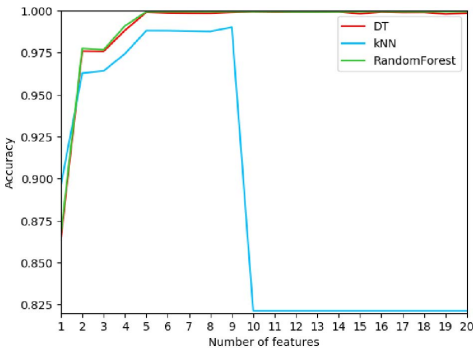


Fig. 8. 3-label accuracy using custom features

V. Conclusion

In this study, we analyzed the feature selection and its impact on the post-hoc local interpretation of the learning model outputs within the context of the botnet detection in IoT networks. A quality metric for the explanations of model decisions reflecting the security analyst perspective is introduced to facilitate this analysis. This metric basically evaluates whether the given explanation describes just only one category or not. If the explanation is prone to explain one category, this is preferable. In our experiments, we utilized a well-known method, LIME, in the post-hoc interpretation phase.

This paper demonstrated that, by using the selected dataset chosen from the problem domain, it is possible to have very high accurate classical machine learning models which can produce explanations that do not create confusion for the security analysts. Our quality metric enabled us to provide such an analysis of interpretability and accuracy in a common picture. Our work is distinguished as it investigates the feature selection and interpretability within the IoT botnet detection domain.

References

[1] I. Andrea, C. Chrysostomou, and G. Hadjichristofi, "Internet of things: Security vulnerabilities and challenges," in *Computers and Communication (ISCC), 2015 IEEE Symposium on*. IEEE, 2015, pp. 180–187.

[2] P. Paganini, "Ovh hosting hit by 1tbps ddos attack, the largest one ever seen," *https://securityaffairs.co/wordpress/51640/cyber-crime/tbps-ddos-attack.html*, 2016.

[3] S. Hilton, "Dyn analysis summary of friday october 21 attack (2016)," *URL https://dyn. com/blog/dyn-analysis-summary-of-fridayoctober-21-attack*, 2016.

[4] B. B. Zarpelão, R. S. Miani, C. T. Kawakani, and S. C. de Alvarenga, "A survey of intrusion detection in internet of things," *Journal of Network and Computer Applications*, vol. 84, pp. 25–37, 2017.

[5] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2016.

[6] C. Zimmerman, "Ten strategies of a world-class cybersecurity operations center," *MITRE corporate communications and public affairs. Appendices*, 2014.

[7] A. Bibal and B. Frénay, "Interpretability of machine learning models and representations: an introduction," in *Proceedings on ESANN*, 2016, pp. 77–82.

[8] A. Vellido, J. D. Martín-Guerrero, and P. J. Lisboa, "Making machine learning models interpretable." in *ESANN*, vol. 12. Citeseer, 2012, pp. 163–172.

[9] A. A. Freitas, "Comprehensible classification models: a position paper," *ACM SIGKDD explorations newsletter*, vol. 15, no. 1, pp. 1–10, 2014.

[10] Z. C. Lipton, "The mythos of model interpretability," *Commun. ACM*, vol. 61, no. 10, pp. 36–43, Sep. 2018. [Online]. Available: http://doi.acm.org/10.1145/3233231

[11] M. T. Ribeiro, S. Singh, and C. Guestrin, "Model-agnostic interpretability of machine learning," *arXiv preprint arXiv:1606.05386*, 2016.

[12] A. Adadi and M. Berrada, "Peeking inside the black-box: A survey on explainable artificial intelligence (xai)," *IEEE Access*, vol. 6, pp. 52 138–52 160, 2018.

[13] M. T. Ribeiro, S. Singh, and C. Guestrin, "Anchors: High-precision model-agnostic explanations," in *AAAI Conference on Artificial Intelligence*, 2018.

[14] R. Guidotti, A. Monreale, S. Ruggieri, D. Pedreschi, F. Turini, and F. Giannotti, "Local rule-based explanations of black box decision systems," *arXiv preprint arXiv:1805.10820*, 2018.

[15] D. Baehrens, T. Schroeter, S. Harmeling, M. Kawanabe, K. Hansen, and K.-R. MÃžller, "How to explain individual classification decisions," *Journal of Machine Learning Research*, vol. 11, no. Jun, pp. 1803–1831, 2010.

[16] P. W. Koh and P. Liang, "Understanding black-box predictions via influence functions," *arXiv preprint arXiv:1703.04730*, 2017.

[17] K. Amarasinghe, K. Kenney, and M. Manic, "Toward explainable deep neural network based anomaly detection," in *2018 11th International Conference on Human System Interaction (HSI)*. IEEE, 2018, pp. 311–317.

[18] J. R. Goodall, E. D. Ragan, C. A. Steed, J. W. Reed, G. D. Richardson, K. M. Huffer, R. A. Bridges, and J. A. Laska, "Situ: Identifying and explaining suspicious behavior in networks," *IEEE transactions on visualization and computer graphics*, vol. 25, no. 1, pp. 204–214, 2019.

[19] D. L. Marino, C. S. Wickramasinghe, and M. Manic, "An adversarial approach for explainable ai in intrusion detection systems," in *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2018, pp. 3237–3243.

[20] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, D. Breitenbacher, A. Shabtai, and Y. Elovici, "N-baiot: Network-based detection of iot botnet attacks using deep autoencoders," *IEEE Pervasive Computing*, vol. 13, no. 9, 2018.

[21] J. Leonard, S. Xu, and R. Sandhu, "A framework for understanding botnets," in *Availability, Reliability and Security, 2009. ARES'09. International Conference on*. IEEE, 2009, pp. 917–922.

[22] C. C. Aggarwal, *Data Mining: The Textbook*. Springer Publishing Company, Incorporated, 2015.

[23] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information." *Psychological review*, vol. 63, no. 2, p. 81, 1956.

Appendix 13

Publication XIII
A. Guerra-Manzanares, J. Medina-Galindo, H. Bahsi, and S. Nõmm. Medbiot: Generation of an iot botnet dataset in a medium-sized iot network. In *Proceedings of the 6th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*, pages 207–218, 2020

MedBIoT: Generation of an IoT Botnet Dataset in a Medium-sized IoT Network

Alejandro Guerra-Manzanares[a], Jorge Medina-Galindo, Hayretdin Bahsi[b] and Sven Nõmm[c]
Department of Software Science, Tallinn University of Technology, Tallinn, Estonia

Keywords: Botnet, Internet of Things, Dataset, Intrusion Detection, Anomaly Detection, IoT.

Abstract: The exponential growth of the Internet of Things in conjunction with the traditional lack of security mechanisms and resource constraints associated with these devices have posed new risks and challenges to security in networks. IoT devices are compromised and used as amplification platforms by cyber-attackers, such as DDoS attacks. Machine learning-based intrusion detection systems aim to overcome network security limitations relying heavily on data quantity and quality. In the case of IoT networks these data are scarce and limited to small-sized networks. This research addresses this issue by providing a labelled behavioral IoT data set, which includes normal and actual botnet malicious network traffic, in a medium-sized IoT network infrastructure (83 IoT devices). Three prominent botnet malware are deployed and data from botnet infection, propagation and communication with C&C stages are collected (Mirai, BashLite and Torii). Binary and multi-class machine learning classification models are run on the acquired data demonstrating the suitability and reliability of the generated data set for machine learning-based botnet detection IDS testing, design and deployment. The generated IoT behavioral data set is released publicly available as MedBIoT data set*.

1 INTRODUCTION

The adoption of the Internet on an increasing wider scope, i.e., providing connectivity capabilities to everyday objects, is a reality. In fact, the rise of the Internet of Things (IoT) has just begun and it is expected to have a major increase in the near future. It was estimated that there would be 26.66 billion active IoT devices by 2019, a figure that may be increased up to 75 billion by 2025 (Statista, 2019). 127 new IoT devices are connected to the Internet every second (McKinsey, 2017) in a wide range of applications, from factories and smart cities sensors to healthcare and car products. The market size is calculated to grow over $212 billion by 2019 and reach $1.6 trillion by 2025 (Liu, 2019). However, the adoption of the IoT technology still poses usability concerns even to early adopters and eager customers, related to device security and data privacy issues (Bosche et al., 2018; Sklavos et al., 2017). Thus, despite its huge growth, the Internet of Things market explosion is still being limited by its main barrier: security (Bertino and Islam, 2017; Bosche et al., 2018; Pratt, 2019).

Their ubiquity will pose a major challenge to security as IoT devices have traditionally lacked of proper control measures and proactive security management (e.g., usage of default passwords, no firmware updates, no access control policy), featuring them as high vulnerable and prone to be compromised devices (Bertino and Islam, 2017). These features have been exploited by malicious actors, being able to compromise the defenseless devices by exploiting its vulnerabilities, gaining remote access and using them as magnification platforms for their massive attacks (Kolias et al., 2017). An IoT botnet is just a particular type of botnet in which the compromised devices are IoT devices, thus showing analogous scheme and dynamics to computer botnets. In this regard, when a vulnerable device is compromised it becomes a *bot*, a member of a larger community of compromised devices, called *botnet*, under the control of a malicious actor, the *botmaster*. The botmaster has remote access and control of the bot over the Internet, without the consent and awareness of the actual owner of the compromised device, using a Command&Control (C&C) server (Silva et al., 2013). Botnets have been used to perpetrate a wide range of malicious attacks, from

[a] https://orcid.org/0000-0002-3655-5804
[b] https://orcid.org/0000-0001-8882-4095
[c] https://orcid.org/0000-0001-5571-1692

*Available at https://cs.taltech.ee/research/data/medbiot

Guerra-Manzanares, A., Medina-Galindo, J., Bahsi, H. and Nõmm, S.
MedBIoT: Generation of an IoT Botnet Dataset in a Medium-sized IoT Network.
DOI: 10.5220/0009187802070218
In *Proceedings of the 6th International Conference on Information Systems Security and Privacy (ICISSP 2020)*, pages 207-218
ISBN: 978-989-758-399-5

massive SPAM and phishing campaigns to distributed denial-of-service (DDoS), the most common usage of a botnet. A DDoS attack targets the availability of online resources, such as websites or services. The main goal is to saturate the targeted server or network with more traffic than it can handle (e.g., receiving an overwhelming amount of messages, connection requests or forged packets) thus provoking the service or website to crash and become unavailable to legitimate users requests (Weisman, 2019).

1.1 Data Sets for IoT Anomaly Detection

The phenomenon of botnet detection in computer networks has been widely studied (Garcia et al., 2014; Feily et al., 2009), with many available data sets at hand (Shiravi et al., 2012), while the most recent IoT network botnet phenomenon has not received the required attention yet, showing a remarkable lack of available data sources.

Data sets for building effective IoT anomaly detection methods rely on the acquisition of both legitimate (normal) and malicious (botnet) behavioral data from IoT networks. Anomaly models are built and trained using only legitimate data to establish the so-called normality patterns. The induced models are tested using legitimate and malicious data, where the metrics related to model's detection performance are evaluated. Therefore, proper and complete data are key components for a high-performance effective intrusion detection system (IDS). Table 1 summarizes the available data sets for IoT anomaly-based intrusion detection systems. As can be observed, a small amount of data sets are available for the specific IoT botnets issue. The available data sets are focused on small-sized IoT networks, reflecting the behavior of a small set of IoT devices. Additionally, a specific and small variety of devices are used (mostly security cameras) limiting the scope of the IoT devices analyzed from the broad domain of available IoT devices. None of the available IoT data sets combine real and emulated devices, which limit the scope of their results to either real or emulated devices. In this regard, our generated data set combines real and emulated devices, using different but common types of IoT devices, not investigated by previous data sets (i.e., fans, locks, light bulbs and switches), in a medium-sized network composed of more than 80 devices. Furthermore, our data set focuses on the first stages of a botnet deployment, such as infection and propagation, while the rest of the data sets focus on the last stages of the botnet lifecycle, mainly detection of attacks (Kirubavathi and Anitha, 2014).

As already stated, the Internet of Things is a reality that will become ubiquitous in the following years. This fact combined with the lack of proper security measures and devices inherent vulnerabilities make IoT devices an easy and appealing target for cyber attackers (Bertino and Islam, 2017). Thus, proper data are in need to create machine learning-based effective detection systems that may help to overcome these limitations. In this regard, there is a remarkable lack of available data sets that might help to build effective IDSs in IoT networks. This research aims to fill this significant gap in IoT anomaly-based IDSs by providing a novel IoT data set obtained from medium size IoT network architecture (more than 80 devices), which includes normal and malicious behavior from different devices (real and emulated) and the deployment of prominent IoT botnets (Mirai, BashLite and Torii). The scale extension enables to capture malware spreading patterns that cannot be seen in small-sized networks, thus providing a more realistic environment. Additionally, this data set includes the behavior of Torii botnet malware which has not been addressed in any other data set before. Finally, this data set provides data for the first stages of botnet deployment (i.e., infection, propagation and communication with C&C server stages), thus complementing the available data sets which mainly focus on attack detection, the main outcome and part of the last stages of the botnet lifecycle (Hachem et al., 2011; Kirubavathi and Anitha, 2014).

This paper is structured as follows: Section 2 provides background information and a review of related literature, Section 3 explains the methodology followed to implement the experimental setup while Section 4 offers a detailed overview of the main outcome of this study, a novel IoT data set for botnet detection, and its verification. Finally, Section 5 concludes the study and highlights its main contributions.

2 LITERATURE REVIEW & BACKGROUND INFORMATION

2.1 Botnets & DDoS Attacks

Botnets have been used to perpetrate record-breaking DDoS attacks. In this regard, in 2016, the journalist Brian Krebs was the target of a record-breaking attack (620 Gbps) to its blog KrebsOnSecurity.com, specifically tailored to take the site offline (Krebs, 2016). A month later, the french hosting provider OVH was attacked by the same botnet (probably BashLite), reaching 1 Tbps and involving over 140.000 compromised

Table 1: Data sets for IoT Anomaly-based IDS.

Data set	Botnet	Number of devices	Device type	Real or Emulated	Network Size	Data set features	Date	Reference
N-Baiot	Mirai BashLite	9	Doorbell Webcam Thermostat Baby monitor Security Camera	Real	Small	115 - statistics	2018	(Meidan et al., 2018a) (Meidan et al., 2018b)
IoT host-based datasets for ID research	Hajime Aidra BashLite Mirai Doflo Tsunami Wroba	2	Multimedia Center Security Camera	Emulated	Small	NA - PCAP & Netflow/Host	2018	(Bezerra et al., 2018a) (Bezerra et al., 2018b)
IoT Network Intrusion Dataset	Mirai	2	Speaker Wi-Fi Camera	Real	Small	NA - PCAP	2019	(Kang et al., 2019)
Bot-IoT	No actual malware - simulated	5	Refrigerator Smart Garage door Weather Monitoring Smart Lights Smart thermostat	Emulated	Small	31+14 - flow	2019	(Moustafa, 2019) (Koroniotis et al., 2019)

cameras/dvr (Pritchard, 2018). The same year, Dyn, a domain name system provider of major websites and services such as CNN, Netflix, Paypal, Visa or Amazon was attacked by the Mirai botnet, using around 100.000 IoT devices and reaching up to 1.2 Tbps, causing the servers to be inoperative and the websites unreachable by the legitimate users for several hours (Weisman, 2019; Hilton, 2016). It is estimated that Dyn lost around 8% of its customers (i.e., 14000 domains) as a consequence of the attack and the lost of trust (Weagle, 2017). This was just the onset for the IoT botnet-based attacks. Since then, the attacks have not stopped, evolving in sophistication and capabilities as the source code of the malware behind the botnets became available to the public (Asokan, 2019). A recent report by F-secure states that cyber-attacks on IoT devices rouse 300% in 2019, reaching the 3 billion attacks, an unprecedented figure (Doffman, 2019). The threat is still alive and growing, caused mainly by the combination of the increase of the number of IoT devices deployed worldwide and the intrinsic vulnerabilities carried by such devices, which can also contain valuable data related to medical or control issues. Nevertheless, one of the major risks is the usage of the IoT endpoints (e.g., a printer or a fridge) as an easy-to-reach and vulnerable entry points to wider and secured networks (Doffman, 2019).

As a result, cyber security for IoT, in the form of early detection of threats, becomes a key issue to detect and mitigate such attacks. In this regard, intrusion detection systems are widely used network security components which aim to detect security threats where preventive security measures are not feasible to implement (Benkhelifa et al., 2018; Sun et al., 2007).

2.2 Intrusion Detection Systems

An intrusion could be defined as a set of activities or actions that compromise one or more components of the IT security model known as CIA triad (i.e., short for Confidentiality, Integrity and Availability) of a specific entity or system. These systems are not restricted to computers, network equipment, firewall, routers or networks but to any information technology system which is under the monitoring scope of an intrusion detection system (IDS) (Sun et al., 2007). Based on that, an intrusion detection system is a security tool that aims to detect and identify the unauthorized individuals willing to break into and misuse a system and also those authorized and legitimate users that abuse of their privileges within the system (Sun et al., 2007). There are four common approaches used for intrusion detection: misuse, anomaly, specification and hybrid (Benkhelifa et al., 2018; Sun et al., 2007; Butun et al., 2013; Zarpelão et al., 2017). They are briefly explained as follows:

- Misuse or signature-based detection systems use known fingerprints or signatures from attacks stored in a database. If an IDS finds a match between the current activities and a known signature it raises the alarm about the detected suspicious behavior. This systems are easily bypassed by not-known or novel attacks, when a signature is not yet available.
- Anomaly-based detection systems are based on the creation of a typical or normal activity profile. Current activities are compared against this normal behavior. If the IDS finds a significant deviation or discrepancies from the normality model it raises the alarm about the suspicious behavior.

These systems success on the detection of novel attacks but they are prone to false positives (i.e., legitimate behavior is detected as malicious behavior) as the normal behavior might not be easy to model, so that being very sensitive to the correctness of the normality model created.

- Specification-based detection systems combine features of misuse and anomaly approaches. They apply anomaly-based principle on set of human generated specifications or constraints about the normal or legitimate behavior. These systems aim to detect novel attacks based on anomalous behavior while reducing the amount of false positives.
- Hybrid detection systems involve the combination of any of the previous approaches, aiming to overcome the weaknesses of one approach using the strengths of another.

One of the most effective and widely used detection methods is the anomaly-based approach, which enables to detect novel attacks but with the inevitable trade-off of being sensitive to the correctness of the generated normality model. In this regard, statistical methods and machine learning algorithms are generally used to generate the normal behavior profile (Zarpelão et al., 2017). Therefore, valid behavioral models should be used in order to obtain the maximum benefit of this approach, depending in a direct manner on the available training data (Bolzoni, 2009). In IoT networks, where a wide variety of devices may coexist in the same network, it is likely to have different normality profiles which emphasizes the need of accurate IoT behavioral data that enable the implementation of effective anomaly-based IDS. Thus, the need of proper data encompassing such differences are highly in demand. However, there is a remarkable lack of available data sets that consider the different network behaviors, devices and architectures that can be found in IoT networks and its major threats. As a result, proper IoT behavioral data are key to train the IDS model for effective intrusion detection in IoT networks.

2.3 Machine Learning-based IDS

Machine learning has shown promising results regarding computer botnet traffic detection (Livadas et al., 2006) and more lately, in the specific IoT botnet detection issue (Zarpelão et al., 2017). As a result of the remarkable increase in IoT related security incidents, researchers have reoriented their focus to deal with the investigation of feasible and effective IoT botnet detection methods involving anomaly-based machine learning approaches. These approaches aim to overcome the intrinsic hardware and software limitations and capabilities of these devices (Zarpelão et al., 2017). In this regard, in Meidan et al. (2018b), Deep Autoencoders, Local Outlier Factor, One-Class Support Vector Machines and Isolation Forest algorithms models built and evaluated using the N-baiot dataset. The results show that all algorithms, except Isolation Forest, effectively detected all Mirai and BashLite simulated attacks. Their proposed method, based on Deep Autoencoders, showed the lowest *false alarms* ratio and required less time to detect the attacks than the other approaches. Prokofiev et al. (2018) used Logistic Regression algorithm to estimate the probability that a device was part of an IoT botnet, focusing on the connection initiation at the propagation stage. Lin et al. (2014) proposed an IoT botnet detection method which combines Support Vector Machines and Artificial Fish Swarm algorithms. McDermott et al. (2018) provided a new application for a text recognition deep learning algorithm (Bidirectional Long Short Term Memory based Recurrent Neural Network), with remarkable success on Mirai botnet attack detection. Doshi et al. (2018), used different network features to train and evaluate the accuracy of *k*-Nearest Neighbors, Support Vector Machines, Decision Tree, Random Forest and Artificial Neural Networks algorithms on the detection Mirai DDoS attacks. A novel IoT malware detection approach using network traffic is proposed in Shire et al. (2019) where Convolutional Neural Networks and binary visualisation technique were used to provide a fast detection method for zero-day malware.

As can be observed, the application of anomaly detection requires the acquisition of malicious traffic which is tested against normal or legitimate traffic in order to evaluate the goodness of the proposed detection model. For this purpose, the data sets should provide both kinds of network traffic in order to assess the effective detection of threats. In this paper we provide demonstrability of the generated data set on classification issues (i.e., supervised learning), for the easiness of interpretation of the results and comparison, but this data set may also be used to build effective anomaly detection models, considered traditionally unsupervised learning.

3 METHODOLOGY

The main outcome of this research is the generation of a labelled behavioral IoT data set, which includes normal and actual botnet malicious network traffic, in a medium-sized IoT infrastructure (composed of more than 80 devices). The focus was on the acquisition

of network data from all the endpoints and servers during the initial propagation of Mirai, BashLite and Torii botnets.

3.1 IoT Network Topology

The network topology created for the purpose of this study is provided in Figure 1. It is composed by 3 connected networks: internet network, monitoring network and IoT LAN network. Their functions and components are described as follows:

- Internet Network: this network is directly connected to the internet in order to provide internet connectivity for the initial configuration of different devices. To restrict the connectivity between networks, a different subnetwork mask is established.
- Monitoring Network: this network provides storage and processing capabilities for the data received from the switch. It is composed by a capture server and a security information and event management (SIEM) server. The capture server is responsible for the collection and storage of the acquired network packages within the whole infrastructure. *Tcpdump* is used to monitor and log the network traffic and store the data in *pcap* file format which is later used as an input by the SIEM server. The SIEM server is a *Splunk* software instance which is responsible for data indexing, filtering, analysis and data set generation (i.e., data processing and labelling).
- IoT LAN Network: this local area network (LAN) allows to spread the malware in a contained manner. This network is composed of physical and virtual IoT devices that generate the behavioral traffic collected by the monitoring network, either benign or malware generated traffic. Virtual devices are deployed using containerization software (i.e., *Docker*). The composition and capabilities of this network devices are explained as follows:
  - Router: this device is responsible for the generation of an isolated network segment allowing only communication between internal devices within this network (i.e., using firewall rules). The router provides IP addresses to this internal devices using Dynamic Host Configuration Protocol (DHCP).
  - Switch: this device is responsible for the acquisition and transfer of the network packages using the *port mirroring* technique. *Port mirroring* is used to clone and transfer network packages that flow through one port to another port, in real time, without affecting the network performance. In this scenario, all devices generated data are captured and transferred to the monitoring network.
  - IoT Management System: this device allows the management of all the IoT devices in a centralized manner. It is deployed using *Hassio* software running on a *Raspberry Pi*, which allows to simulate the same network behavior of real implementations. In this network, 4 different IoT devices were emulated: fan, lock, light bulb and switch. Each device allows the remote control of different features. For instance, the fan allows the selection of speed, oscillation state, current fan state and turning on/off capabilities.
  - Virtual IoT Devices: this device allows the virtualization of IoT devices using *Docker* containers. It is deployed using a *Raspberry Pi* which allows to emulate the behavior of an IoT device.
  - Wireless Access Point: this device allows network connection to the non-ethernet compatible devices. It is configured to allow the router the capability of assigning IP addresses (via DHCP), thus avoiding the possibility of IP address duplicates.
  - BashLite C&C Server: this server is the command and control unit of the BashLite botnet. FTP and web services are installed to allow the spreading of the malware. The server is also used to compile the malware binaries used to propagate the infection.
  - Mirai C&C Server: this server is the command and control device of the Mirai botnet. FTP and web services are installed to allow the malware propagation. The server is also used to compile the malware binaries used to spread the infection.
  - DNS Server Sinkhole: this server provides the domain name resolution for the Mirai botnet. It is also used as a sinkhole for the domains that Torii malware requests connection. The sinkhole avoids the actual connection between Torii and the domain of its C&C server, providing effective malware contention.
  - Physical Devices: this devices compose the collection of real IoT devices of this network. It is composed by 3 different devices: Sonoff tasmota smart switch, TpLink smart switch and TpLink smart bulb. All of them allow external device management and provide different features. For instance, the light bulb allows to

control light intensity, status and turn on/off capabilities.

In order to create a medium-sized network, virtual devices are created and physical devices deployed, summing up a total amount of 83 devices. Table 2 shows the composition of the IoT LAN network. As can be observed, 80 devices are emulated and 3 are actual physical devices. The virtual devices have ARM architecture as it is inherited from the *Raspberry Pi* while the physical devices have MIPS architecture. This fact conditions the malware binary used to infect the device, being architecture-dependant, and enriches the spectrum of the data, considering a wider variety of IoT devices. The *features* column provides outlines the actions that the deployed IoT devices are capable to perform.

Table 2: IoT network device composition.

Device	Type	Features	Architecture	Number of devices
Switch	Physical	Turn On Turn Off	MIPS	2
Light bulb	Physical	Turn On Turn Off Intensity	MIPS	1
Lock	Virtual	Lock Unlock	ARM	20
Fan	Virtual	Turn On Turn Off Speed Oscillation	ARM	20
Switch	Virtual	Turn On Turn Off	ARM	20
Light bulb	Virtual	Turn On Turn Off Intensity	ARM	20

3.2 IoT Behavior

The simulation of devices' behavior can be performed in several ways, ranging from the imitation of the behavior by manual usage of the devices to the automation of the execution of specific functions/tasks using *scripts*. The quality and consistency of the simulated behavior is key to create a high quality data set that provide realistic data input for effective IDS solutions. In such cases, the acquisition of real and relevant data regarding the normal usage patterns provide a realistic baseline for the simulation of the behavior. For instance, in a normal living room, the research showed that a light bulb had a mean usage of 1.7h per day while this value achieved 2.3h in the case of a light bulb in the kitchen (Gifford et al., 2012). This information provided a baseline for the simulation of benign behavior in our experimental setup. In the case of malware behavior it is simulated by the execution of the different modules within the botnet, providing a real output of the botnet behavior.

3.2.1 Legitimate Behavior

An automated execution approach is utilised for the simulation of benign behavior. This approach takes into account the architecture of the device, as stated in Table 2, performed using a python script and MQTT (MQ Telemetry Transport) protocol, which is a communication protocol used to control IoT devices. The IoT management system allows to automate this control and perform scheduled tasks on connected IoT devices. A *script* with trigger actions is configured and deployed. In this scenario, the following legitimate behavior is simulated using the following triggers:

- All devices are turned on at 8.00 AM
- Each time a device state changes, the management system starts a countdown until the next state change.
- The countdown value is randomized.
- The maximum limit of changes is established in 20 and a maximum of 3h on *ON* state is set.
- All devices are turned off at 07.00 PM
- In order to simulate a working environment, execution of the triggers is limited from Monday to Friday.

By the usage of the previous triggers, network packages are generated along the network, captured and stored. The captured network packages provide the following communication information: time, protocol used, TCP stream, TCP stream size, source IP, destination IP, MAC addresses, TCP raw message and response code.

3.2.2 Malicious Behavior

The malicious behavior is generated by the deployment of three prominent botnet malware within the controlled environment: Mirai (Antonakakis et al., 2017), BashLite (Marzano et al., 2018) and Torii (Kroustek et al., 2018). Mirai and BashLite botnets have been widely studied and malware source code is available on the Internet. Thus its deployment is fully controlled in the lab environment using a C&C server for each botnet and the source code is modified to connect with this specific C&C server. Torii source code is not yet available on the Internet, thus the samples used for its deployment within the controlled environment were obtained from Hybrid Analysis archive (Crowdstrike, 2019). In order to avoid Torii malware to connect with its actual C&C server, special contention measures are in place. Mirai, BashLite and Torii botnet propagation is performed and controlled

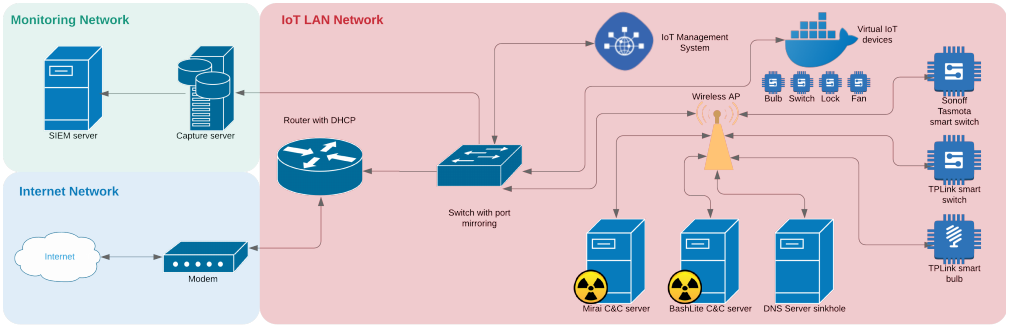


Figure 1: Medium-sized IoT network topology.

within the environment using different strategies, explained in the following paragraphs.

- **Botnet Propagation Techniques**
  - Mirai and *Yakuza* version of BashLite are configured and executed within the controlled environment, modifying the malware source codes in order to link the infection binaries with the corresponding in-lab C&C servers. Once the botnet is properly set, droppers are the medium used by these botnets to download and install the appropiate malware file according to the victim's architecture which once executed will run the bot daemon, compromising the device successfully.
  - Torii malware has not been profoundly studied yet, so the deployment of this malware in the lab environment carries further risks. In order to contain and eliminate the risks of improper use of the devices by Torii's actual botmaster, a sinkhole in the DNS server and firewall rules are used. Torii connection attempts with its C&C server are permanently denied and redirected. As a result of the lack of proper knowledge of Torii malware spreading methods and source codes, the binaries are deployed manually within the lab environment. The obtained sample is specifically tailored to target ARM devices. The malware is executed running the executable with root privileges in the target devices, allowing to spread the malware through the IoT devices.
- **Botnet Contention Methods**
  - One of the major risks within the lab is the abuse of the devices by real attackers. In this regard, Torii poses a major challenge. Contrarily to Mirai and BashLite, Torii has not been deeply analyzed and poses a risk within the lab environment that has to be addressed. Unsuccessful botnet contention may lead to unauthorized usage of the IoT devices by real attackers to perpetrate attacks or collect rellevant data. Two major risks are found within this experimental setup which are addressed and outlined as follows:
    1. Possibility of existence of hidden code in Mirai's source code to connect to the real C&C server
    2. Torii's unknown spread techniques and functionalities

    Even though Mirai spreading techniques are well-known, additional security measures are taken to ensure effective contention of the malware. To address this issues, a sinkhole and firewall rules are in place to deny possible connection attemps to the real C&C servers. The DNS sinkhole redirects the connection attempts by resolving the name resolution request with a controlled IP address. Firewall rules are set in the router to block/control the traffic based on known network masks.

Botnet malware are deployed at different times within 6 days (i.e., each let run free for 2 consecutive days) aiming to obtain relevant botnet information and eliminate undesired overlapping of information. Furthermore, Mirai malware is capable of detect malware running on a specific device and remove it in order to take the single control of it. A limited number of devices are infected in each botnet deployment. In the case of BashLite malware, 40 devices were infected, chosen in a pseudo-randomized way by limiting the scope of devices scanned and infected. Mirai botnet malware infected 25 devices, limited by the change of configuration to restrict the internal scanner to spread within the lab IP ranges. Torii botnet malware was manually deployed in 12 devices, all under the controlled scope of the DNS sinkhole.

3.3 IoT Behavior Verification

In order to verify the suitability of the IoT behavioral data set generated within the experimental setup for detection purposes, the generated data are further processed and used to build and test machine learning-based classification models. Machine learning classification models aim to correctly predict the label or category of an unknown data point based on features (also called predictors) found on the training data provided during the model training/building phase (i.e., supervised learning). Binary classification is used when the data points are split into to two mutually exclusive categories (e.g., benign and malware) while multi-class classification deals when more than two categories are present within the data (e.g., benign, Mirai, BashLite and Torii). In order to validate the outcome of the experimental setup, both approaches are used, inducing binary and multi-class machine learning classification models, which are validated using *k*-fold cross validation.

From the source *pcap* files captured within the lab, features are extracted and used as predictors/input for the machine learning models. The features used in this lab are computed as in Mirsky et al. (2018). A total of 100 network traffic statistical features are calculated, within different time windows. Table 3 provides a brief description of the generated features. As can be observed, statistical features are calculated in relation to 4 major categories for each of the 5 time windows (i.e, 100ms, 500ms, 1.5s, 10s and 1min).

Table 3: Feature Categories.

Categories	Features
Host-MAC&IP	Packet count, mean and variance
Channel	Packet count, mean, variance, magnitude, radius, covariance, and correlation
Network Jitter	Packet count, mean and variance of packet jitter in channel
Socket	Packet count, mean, variance, magnitude, radius, covariance and correlation

After the features are extracted, a random sample of data points are selected for each class and used to train/test machine learning models using 10-fold cross validation. Four traditional machine learning algorithms are used to induce and test classifier models. The main objective for these tests is to demonstrate the suitability of the present data set for machine learning-based anomaly and classification detection models. In this regard, there is no model hyper-parameter optimization performed on the induced models. Default *scikit_learn* library (version 0.20) configurations are used, leaving room for improvement on the classifiers performance. In this relation, *k*-Nearest Neighbors (*k*-NN), Support Vector Machines (SVM), Decision Tree (DT) and Random Forest (RF) algorithms are implemented. For each of the models, four performance metrics are reported: accuracy, precision, recall and F_1 score. They are briefly described as follows:

- Accuracy: ratio of the correctly classified test instances among all test instances.
- Precision: fraction of positive instances correctly classified among all the positive classified instances.
- Recall: fraction of positive instances correctly classified among all the actual positive instances.
- F_1 score: harmonic mean of precision and recall metrics.

All the performance metrics are bounded on the interval [0, 1]. In general, a value close to 1 may be deemed as a positive or good result for the given task while a value close to 0 as a bad performance. In this regard, for classification tasks, the higher the value the better the classifier performance on label detection and discrimination, thus inferring that the data and the classifier are suitable for that purpose. In our specific case, if the classifiers show a performance close to 1 in all metrics it may be inferred that the data is suitable for machine learning-based IoT botnet detection and that the data labels (e.g., legitimate and malware) can be discriminated effectively.

4 RESULTS

4.1 IoT Behavioral Data Set

The network packets collected in the IoT LAN network are redirected to the monitoring network using the port mirroring technique, where the SIEM software (i.e., *Splunk*) was used to process and label the data, thus allowing to create the final data set. This final data set is generated in two versions: structured (features are computed and extracted from the raw data) and non-structured format (raw *pcap* files). The total number of packets captured during the experimental setup are provided in Table 4.

Table 4: Network data captured.

Number of packets	Traffic type	Number of devices
4,143,276	BashLite	40
842,674	Mirai	25
319,139	Torii	12
12,540,478	Benign	83

As can be observed, a total amount of 17,845,567 network packets were captured within the experimental setup. Around 30% of this traffic was deemed and labelled as malicious while 70% corresponds to legitimate network traffic. Using *Splunk* it is possible to analyze and provide further details about the type of communication. Regarding the legitimate network traffic, 32% of the packages are found to be related to system updates, 53% to device communication (MQTT protocol) and 15% to other network data (e.g., TLS errors, pings, etc). Mirai and BashLite source codes are configured to convey different kind of communications on different ports, with the purpose of facilitating the posterior analysis of the data. In this relation, malicious traffic analysis shows that 68% of the data captured is related to the malware propagation activity while 32% to the communication between the C&C servers and bots. In the case of Torii, malicious traffic only includes data regarding the initial infection of the device as the containment measures did not allow the real C&C to reach the device and trigger posterior botnet events such as propagation. The generated data set is made publicly available in the following url: https://cs.taltech.ee/research/data/medbiot

4.2 IoT Behavior Verification

4.2.1 Binary Classification

Binary or two-class classification models are induced and 10-fold cross validated for four widely used machine learning classification models. In this case, the data is divided in two classes or labels: legitimate and malware (mixed data from the three malware subclasses). More specifically, the data set used is created by random selection of 3000 data points from the legitimate traffic, thus conforming the legitimate class data. The malware class is composed of 1000 random selected data points for each one of the malware botnets deployed within the lab, summing up to 3000 data points for this class. As a result, a balanced data set is created and used to perform the binary classification task. Support Vector Machines algorithm showed a poor performance in all assessed metrics, thus is not reported in the results, which are provided in Table 5.

Table 5: Binary classification results.

Model	Accuracy	Precision	Recall	F_1 score
k-NN	0.9025	0.9082	0.9025	0.9001
DT	0.9315	0.9448	0.9315	0.9293
RF	0.9532	0.9580	0.9532	0.9481

As can be observed, Random Forest algorithm is able to discriminate over 95% of the data points, thus detecting effectively the vast majority of the malware traffic. Decision Tree and *k*-NN show slightly less discriminatory performance, but over 90% in all performance metrics in both cases. The malware traffic, which is composed of a mixture of 3 different botnet malware, is effectively discriminated from legitimate traffic, as can be confirmed by the normalized confusion matrix provided in Table 6, extracted from a Random Forest model. As already stated, SVM showed bad performance, and its results are not reported. Nevertheless, this fact may suggest that the data is not linearly separable, thus being SVM a not suitable classifier model for this task unlike the other algorithms used. These results emphasize the effective capabilities of machine learning approaches to detect botnet malware traffic, even in the first stages of its deployment (i.e., infection, propagation and communication with the C&C server stages) and disregarding the botnet malware employed. Furthermore, the data set created within this lab demonstrates its suitability to be used as a medium-sized realistic IoT data set for IoT botnet detection scenarios and IDS testing.

Table 6: Confusion matrix of RF binary classification.

		Predicted	
		Malware	Legitimate
Actual	Malware	291	9
	Legitimate	7	293

4.2.2 Multi-class Classification

In this setting, multi-class classification models are induced and 10-fold cross validated for the same algorithms employed in the binary approach. In this case, the data was divided in four classes or labels: legitimate, Mirai, BashLite and Torii. The data set used is created by random selection of 2000 data points of each of the possible classes, summing up to 8000 data points, evenly distributed in 4 labels. The main aim of this configuration is not only to test the legitimate/malware discrimination, as in the binary approach, but also the discrimination of the specific malware source. As in the previous setting, Support Vector Machines algorithm showed a poor performance in all metrics, thus its performance is not reported. Table 7 provides the results for the multi-class classification task.

As can be seen, Random Forest model outperforms Decision Tree and *k*-NN algorithms in the multi-class classification task, in a similar fashion as in the binary models. More specifically, RF algorithm

Table 7: Multi-class classification results.

Model	Accuracy	Precision	Recall	F_1 score
k-NN	0.8706	0.8849	0.8706	0.8505
DT	0.9516	0.9584	0.9516	0.9499
RF	0.9766	0.9824	0.9766	0.9657

is able to discriminate more accurately the labels in the multi-class scenario than in the binary setting, being able to discriminate accurately over 97% of the data points. As shown in Table 8, extracted from the Random Forest model, the classification model is very accurate in all cases, not showing any significant bias towards any of the possible labels. The results obtained suggest that the source of network traffic can be effectively discriminated in earlier stages of botnet infection. They also demonstrate that the learning capabilities of machine learning-based detection can be accurate not only in the binary setting but also in the specific discrimination of different sources of malicious traffic in medium-sized IoT networks.

Table 8: Confusion matrix of RF multi-class classification.

		Predicted			
		Mirai	BashLite	Torii	Legitimate
Actual	Mirai	197	0	0	3
	BashLite	2	196	0	2
	Torii	0	0	198	2
	Legitimate	2	0	0	198

5 CONCLUSIONS

The exponential growth of the Internet of Things is a fact and these devices will become ubiquitous in the near future. The increasing connectivity capabilities of these devices in conjunction with their traditional lack of security features make them an appealing target for cyber-attackers. Malicious actors compromise the vulnerable IoT devices and use them as an amplification platform of their attacks, becoming part of the so-called *botnet*. Botnets have been extensively used to deliver massive spam campaigns and perpetrate record-breaking DDoS attacks that may lead to nefarious consequences. Therefore, there is an increasing need to overcome the lack of security of these devices. The proposed solutions are mainly coming from machine learning-based approaches.

The performance of machine learning algorithms heavily rely on data quality and quantity. In this relation, there is a remarkable lack of data sources in the specific IoT networks scenario. The experimental setup of this research aims to fulfill this gap by providing a novel data set with network data collected from a medium-sized IoT network architecture, which is composed of legitimate and botnet malware traffic. Three IoT botnet malware are deployed in real and emulated IoT devices and data are acquired from the first stages of botnet deployment, such as infection, propagation and communication with C&C server. These data complements the already existing data sets which mainly focus on detection of botnet attacks, part of the last stages of a botnet deployment. In this sense, by focusing on early stages of botnet deployment, the proposed data set provides the opportunity to perform early detection of the threat, previous to the perpetration of an attack, being able to prevent such attacks and botnet growth. Three prominent botnet malware are deployed in this research, one of them is a complete novelty (i.e., Torii), not being deployed before in any other available data set. The other two are well-known IoT botnet malware whose source code is publicly available and have been used in other data sets (i.e., Mirai and BashLite). The currently available data sets, summarized in Table 1, focus on small-sized networks (usually less than 10 devices), using either emulated or real devices, thus providing limited interactions between devices inside the network. The generated data set addresses these limitations by combining emulated and real devices to create a medium-sized network (i.e., 83 devices). A larger network size may provide different insights and interactions than smaller IoT networks. Finally, machine learning models are built and validated using this data to demonstrate the suitability of this data set as a reliable data source for botnet detection in general and IDS testing and deployment in particular. The data set generated within the experimental setup is made publicly available, aiming to overcome the scarcity of relevant data sources in IoT network security and limitations of the existing data sets.

REFERENCES

Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., et al. (2017). Understanding the mirai botnet. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1093–1110.

Asokan, A. (2019). Massive botnet attack used more than 400,000 iot devices. Retrieved from: https://www.bankinfosecurity.com/massive-botnet-attack-used-more-than-400000-iot-devices-a-12841.

Benkhelifa, E., Welsh, T., and Hamouda, W. (2018). A critical review of practices and challenges in intrusion detection systems for iot: Toward universal and resilient systems. *IEEE Communications Surveys & Tutorials*, 20(4):3496–3509.

Bertino, E. and Islam, N. (2017). Botnets and internet of things security. *Computer*, (2):76–79.

Bezerra, V. H., da Costa, V. G. T., Martins, R. A., Junior, S. B., Miani, R. S., and Zarpelao, B. B. (2018a). Data set. http://www.uel.br/grupo-pesquisa/secmq/dataset-iot-security.html.

Bezerra, V. H., da Costa, V. G. T., Martins, R. A., Junior, S. B., Miani, R. S., and Zarpelao, B. B. (2018b). Providing iot host-based datasets for intrusion detection research. In *Anais do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 15–28. SBC.

Bolzoni, D. (2009). *Revisiting Anomaly-based Network Intrusion Detection Systems.* University of Twente, Enschede, Netherlands.

Bosche, A., Crawford, D., Jackson, D., Schallehn, M., and Schorling, C. (2018). Unlocking opportunities in the internet of things. Retrieved from: https://www.bain.com/contentassets/5aa3a678438846289af59f62e62a3456/bain_brief_unlocking_opportunities_in_the_internet_of_things.pdf.

Butun, I., Morgera, S. D., and Sankar, R. (2013). A survey of intrusion detection systems in wireless sensor networks. *IEEE communications surveys & tutorials*, 16(1):266–282.

Crowdstrike (2019). Hybrid analysis. Retrieved from: https://www.hybrid-analysis.com/.

Doffman, Z. (2019). Cyberattacks on iot devices surge 300% in 2019, 'measured in billions', report claims. Retrieved from: https://www.forbes.com/sites/zakdoffman/2019/09/14/dangerous-cyberattacks-on-iot-devices-up-300-in-2019-now-rampant-report-claims/#574229995892.

Doshi, R., Apthorpe, N., and Feamster, N. (2018). Machine learning ddos detection for consumer internet of things devices. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 29–35. IEEE.

Feily, M., Shahrestani, A., and Ramadass, S. (2009). A survey of botnet and botnet detection. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 268–273. IEEE.

Garcia, S., Grill, M., Stiborek, J., and Zunino, A. (2014). An empirical comparison of botnet detection methods. *computers & security*, 45:100–123.

Gifford, W. R., Goldberg, M. L., Tanimoto, P. M., Celnicker, D. R., and Poplawski, M. E. (2012). Residential lighting end-use consumption study: Estimation framework and initial estimates. Retrieved from: https://www1.eere.energy.gov/buildings/publications/pdfs/ssl/2012_residential-lighting-study.pdf.

Hachem, N., Mustapha, Y. B., Granadillo, G. G., and Debar, H. (2011). Botnets: lifecycle and taxonomy. In *2011 Conference on Network and Information Systems Security*, pages 1–8. IEEE.

Hilton, S. (2016). Dyn analysis summary of friday october 21 attack. Retrieved from: https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/.

Kang, H., Ahn, D. H., Lee, G. M., Yoo, J. D., Park, K. H., and Kim, H. K. (2019). Iot network intrusion dataset. http://dx.doi.org/10.21227/q70p-q449.

Kirubavathi, G. and Anitha, R. (2014). Botnets: A study and analysis. In *Computational Intelligence, Cyber Security and Computational Models*, pages 203–214. Springer.

Kolias, C., Kambourakis, G., Stavrou, A., and Voas, J. (2017). Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84.

Koroniotis, N., Moustafa, N., Sitnikova, E., and Turnbull, B. (2019). Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset. *Future Generation Computer Systems*, 100:779–796.

Krebs, B. (2016). Krebsonsecurity hit with record ddos. Retrieved from: https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/.

Kroustek, J., Iliushin, V., Shirokova, A., Neduchal, J., and Hron, M. (2018). Torii botnet - not another mirai variant. Retrieved from: https://blog.avast.com/new-torii-botnet-threat-research.

Lin, K.-C., Chen, S.-Y., and Hung, J. C. (2014). Botnet detection using support vector machines with artificial fish swarm algorithm. *Journal of Applied Mathematics*, 2014.

Liu, S. (2019). Global iot market size 2017-2025. Retrieved from: https://www.statista.com/statistics/976313/global-iot-market-size/.

Livadas, C., Walsh, R., Lapsley, D. E., and Strayer, W. T. (2006). Using machine learning techniques to identify botnet traffic. In *LCN*, pages 967–974. Citeseer.

Marzano, A., Alexander, D., Fonseca, O., Fazzion, E., Hoepers, C., Steding-Jessen, K., Chaves, M. H., Cunha, Í., Guedes, D., and Meira, W. (2018). The evolution of bashlite and mirai iot botnets. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00813–00818. IEEE.

McDermott, C. D., Majdani, F., and Petrovski, A. V. (2018). Botnet detection in the internet of things using deep learning approaches. In *2018 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

McKinsey (2017). What's new with the internet of things? Retrieved from: https://www.mckinsey.com/industries/semiconductors/our-insights/whats-new-with-the-internet-of-things.

Meidan, Y., Bohadana, M., Mathov, Y., Mirsky, Y., Breitenbacher, D., Shabtai, A., and Elovici, Y. (2018a). detection_of_iot_botnet_attacks_n_baiot data set. http://archive.ics.uci.edu/ml/datasets/detection_of_IoT_botnet_attacks_N_BaIoT.

Meidan, Y., Bohadana, M., Mathov, Y., Mirsky, Y., Breitenbacher, D., Shabtai, A., and Elovici, Y. (2018b). N-baiot—network-based detection of iot botnet attacks using deep autoencoders. *IEEE Pervasive Computing*, 17(3):12–22.

Mirsky, Y., Doitshman, T., Elovici, Y., and Shabtai, A. (2018). Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*.

Moustafa, N. (2019). The bot-iot dataset. http://dx.doi.org/10.21227/r7v2-x988.

Pratt, M. K. (2019). Top challenges of iot adoption in the enterprise. Retrieved from: https://internetofthingsagenda.techtarget.com/feature/Top-challenges-of-IoT-adoption-in-the-enterprise.

Pritchard, M. (2018). Ddos attack timeline: Time to take ddos seriously. Retrieved from: https://activereach.net/newsroom/blog/time-to-take-ddos-seriously-a-recent-timeline-of-events/.

Prokofiev, A. O., Smirnova, Y. S., and Surov, V. A. (2018). A method to detect internet of things botnets. In *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, pages 105–108. IEEE.

Shiravi, A., Shiravi, H., Tavallaee, M., and Ghorbani, A. A. (2012). Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *computers & security*, 31(3):357–374.

Shire, R., Shiaeles, S., Bendiab, K., Ghita, B., and Kolokotronis, N. (2019). Malware squid: A novel iot malware traffic analysis framework using convolutional neural network and binary visualisation. In *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, pages 65–76. Springer.

Silva, S. S., Silva, R. M., Pinto, R. C., and Salles, R. M. (2013). Botnets: A survey. *Computer Networks*, 57(2):378–403.

Sklavos, N., Zaharakis, I. D., Kameas, A., and Kalapodi, A. (2017). Security & trusted devices in the context of internet of things (iot). In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 502–509. IEEE.

Statista (2019). Internet of things - number of connected devices worldwide 2015-2025. Retrieved from: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.

Sun, B., Osborne, L., Xiao, Y., and Guizani, S. (2007). Intrusion detection techniques in mobile ad hoc and wireless sensor networks. *IEEE Wireless Communications*, 14(5):56–63.

Weagle, S. (2017). Financial impact of mirai ddos attack on dyn revealed in new data. Retrieved from: https://www.corero.com/blog/797-financial-impact-of-mirai-ddos-attack-on-dyn-revealed-in-new-data.html.

Weisman, S. (2019). Emerging threats - what is a distributed denial of service attack (ddos) and what can you do about them? Retrieved from: https://us.norton.com/internetsecurity-emerging-threats-what-is-a-ddos-attack-30sectech-by-norton.html.

Zarpelão, B. B., Miani, R. S., Kawakani, C. T., and de Alvarenga, S. C. (2017). A survey of intrusion detection in internet of things. *Journal of Network and Computer Applications*, 84:25–37.

Appendix 14

Publication XIV
A. Guerra-Manzanares, J. Medina-Galindo, H. Bahsi, and S. Nõmm. Using medbiot dataset to build effective machine learning-based iot botnet detection systems. In *International Conference on Information Systems Security and Privacy*, pages 222–243. Springer, 2020

Using MedBIoT Dataset to Build Effective Machine Learning-Based IoT Botnet Detection Systems

Alejandro Guerra-Manzanares( ), Jorge Medina-Galindo, Hayretdin Bahsi, and Sven Nõmm

Department of Software Science, Tallinn University of Technology, Tallinn, Estonia
{alejandro.guerra,hayretdin.bahsi,sven.nomm}@taltech.ee

Abstract. The exponential increase in the adoption of the Internet of Things (IoT) technology combined with the usual lack of security measures carried by such devices have brought up new risks and security challenges to networks. IoT devices are prone to be easily compromised and used as magnification platforms for record-breaking cyber-attacks (i.e., Distributed Denial-of-Service attacks). Intrusion detection systems based on machine learning aim to detect such threats effectively, overcoming the security limitations on networks. In this regard, data quantity and quality is key to build effective detection models. These data are scarce and limited to small-sized networks for IoT environments. This research addresses this gap generating a labelled behavioral IoT data set, composed of normal and actual botnet network traffic in a medium-sized IoT network (up to 83 devices). Mirai, BashLite and Torii real botnet malware are deployed and data from early stages of botnet deployment is acquired (i.e., infection, propagation and communication with C&C stages). Supervised (i.e. classification) and unsupervised (i.e., anomaly detection) machine learning models are built with the data acquired as a demonstration of the suitability and reliability of the collected data set for effective machine learning-based botnet detection intrusion detection systems (i.e., testing, design and deployment). The IoT behavioral data set is released, being publicly available as MedBIoT data set.

Keywords: Botnet · Internet of Things · Dataset · Intrusion detection · Anomaly detection · IoT · Machine learning

1 Introduction

The inter-connectivity of nowadays world's elements is a fact. Internet has extended the connectivity and communication capabilities like never before, not only to humans but also for everyday objects. Now it is possible to interact and control via Internet objects such as TV's, refrigerators, light bulbs or thermostats. The so-called Internet of Things (shortened as IoT) has just started its expansion, expecting a major growth in the near future. It was estimated that there were around 22 billion connected IoT devices by 2018, a figure expected to reach 50 billion by 2030 [52]. Globally, 127 new IoT devices are connected to the Internet every second [34] encompassing a wide range of

S. Furnell et al. (Eds.): ICISSP 2020, CCIS 1545, pp. 222–243, 2022.
https://doi.org/10.1007/978-3-030-94900-6_11

applications from healthcare and manufacturing to automotive and agriculture. A typical consumer owns an average of four IoT devices that communicate directly with the cloud [34]. The global IoT market size is estimated to grow over $248 billion by 2020 and reach the $1.6 trillion figure by 2025 [51]. In spite of its wide spread and significant growth, the IoT technology still poses concerns even to the early adopters and eager customers, mostly related to security and data privacy [10,34,50]. IoT devices have been identified as potential entry points and enticing targets for cyberattacks, exposing their vulnerabilities and facing challenges for their massive adoption [34]. Thus, despite its vast growth, the Internet of Things market blast is still constrained by its main barrier: security [5,10,42].

The ubiquity of IoT devices might pose a major challenge to security as IoT devices have traditionally lacked of proper control measures, maintenance and proactive security management (e.g., usage of default passwords, no firmware updates, no access control policy), featuring them as highly vulnerable and easy to be compromised devices [5,34]. These weaknesses have been exploited by attackers, being able to compromise the defenseless devices by exploiting its vulnerabilities, thus gaining remote control and using them as amplification platforms for their massive disruptive attacks [25].

Effective IoT botnet attack anomaly detection methods rely on the usage of appropriate data. These data sets are characterized by the collection of normal (legitimate) and malicious (botnet) behavioral data from IoT networks. Anomaly detection models are built on the basis of legitimate data, establishing a normality pattern. The induced models are assessed using normal and malicious data. Performance metrics are computed and used to evaluate the model's detection capabilities. Therefore, accurate and complete data are key elements to build highly effective intrusion detection systems (IDS).

In this regard, as can be observed in Table 1, all the available data sets focus on small-sized IoT networks and on a specific and small variety of devices, mostly cameras. As a result, the behavior of a small set of devices is acquired, considerably limiting the scope of the IoT devices analyzed from the vast and varied domain of the existing IoT devices. Furthermore, none of the data sets use a combination of real and emulated IoT devices, which impacts and limits the scope of their results to either real or emulated devices.

This research aims to fill this substantial gap by providing a novel IoT data set acquired from a medium size IoT network architecture (i.e., 83 devices), including normal and malicious behavioral traffic from both real and emulated devices and the deployment of three prominent IoT botnets (i.e., Mirai, BashLite and Torii). The size extension allows to capture malware spreading patterns and interactions that cannot be observed in small-sized networks, providing a more realistic environment. Furthermore, no data set uses the combination of emulated and real devices within the same network. Additionally, this data set includes the behavior of Torii botnet malware, being the first publicly available data set to deploy it. Lastly, this data set provides and focuses on malware infection, propagation and communication with C&C server phases, the first stages of actual botnet deployment, while the other data sets focus on the last stages of the botnet life cycle, the attack phase [29]. In this relation, this data set can be seen as a complement of the already available data sets, which mainly focus on attack detection, the main outcome and part of the later stages of the botnet life cycle [22,29].

This paper is an extension of the original paper [21] which presented the novel MedBIoT data set. This paper builds up on the original paper by adding more detailed analysis and comparison of publicly released data sets for IoT botnet detection (Sect. 2.4), recently published research literature on the field (Sect. 2.3) and, more significantly, extends the experimentation performed with MedBIoT data set to anomaly-based detection models (Sect. 4.2, anomaly detection). In this regard, while the original paper focused on supervised machine learning (i.e., classification) this paper provides tests and experimentation using unsupervised machine learning models (i.e., anomaly detection) which show and emphasize the goodness of the data to build any kind of effective machine learning-based intrusion detection system. The data set is available at https://cs.taltech.ee/research/data/medbiot/.

The paper structure is as follows: background information and literature review are provided in Sect. 2, while Sect. 3 explains the methodology implemented in the experimental setup. Section 4 shows a comprehensive overview of the main outcome of this research, a novel IoT botnet data set, and its verification. Lastly, Sect. 5 wraps up the study and highlights its major contributions.

2 Background Information

2.1 Botnets and DDoS Attacks

An IoT botnet is a specific type of computer botnet in which the compromised devices are IoT devices, thus presenting analogous schemes and dynamics as computer botnets. In this regard, when a device has its vulnerabilities exploited, thus being compromised, it becomes a *bot*. Bots are grouped on a large community of compromised devices, called *botnet*. A botnet is typically under the control of a malicious actor, the *botmaster*. The botmaster controls remotely the bot over the Internet, using Command & Control (C&C) servers [49]. This privileged access is unauthorized, there is no consent or awareness from the real owner of the compromised device.

IoT Botnets are used to perpetrate a wide scope of attacks, from massive SPAM and phishing campaigns to distributed denial-of-service (DDoS), the most common attack performed using botnets. A DDoS attack aims to compromise the availability of online resources, such as websites or services. This goal is achieved overloading the targeted server or network with more traffic than it can handle (e.g., sending an overwhelming amount of messages, connection requests or forged packets) and provoking the service or website to get saturated and crashing. As a result, the crashed machine becomes unavailable and unresponsive to the legitimate users requests [56]. In this regard, KrebsOnSecurity.com, the blog of the journalist Brian Krebs, was the target of a record-breaking attack (i.e., 620 Gpbs) in 2016. The attack, performed using Mirai botnet, was specifically tailored to tackle the site down [27]. Just a month later, the company OVH, a well-known hosting provider, was attacked by BashLite botnet hitting 1 Tbps and involving over 140.000 compromised cameras/dvr [43]. The same year, Dyn, a domain name system provider of well-known websites and services such as Netflix, PayPal, Visa, CNN and Amazon was attacked by 100.000 IoT devices belonging to Mirai botnet. The attack reached up to 1.2 Tbps, disrupting the services and causing the servers to be inoperative and the websites unavailable for several hours [23,56].

As a collateral damage from the attack and the loss of trust, it is estimated that Dyn lost around 8% of its customers (i.e., 14000 domains) [55]. And these attacks were just the beginning. Since then, IoT botnet-based attacks have not stopped. On the contrary, they have evolved in sophistication and capabilities, influenced by the public release of the source code behind some prominent botnet malware [2]. According to F-secure, in 2019, cyber attacks on IoT devices rouse 300%, reaching the unprecedented figure of 3 billion attacks [14]. Therefore, the threat is still alive and growing, mainly caused by the conjunction of factors such as the increase of the number of devices deployed worldwide and the inherent vulnerabilities that characterise them, which also poses at risk the data they carry and store, usually in an unencrypted manner [40], which might be deemed as confidential and related to medical or control issues in many applications. Nevertheless, one of the major threats is the leveraging of IoT endpoints, such as printers or fridges, as highly vulnerable entry points to wider and otherwise considered to be secure networks [14].

Consequently, cyber security for the IoT domain, in the form of early detection of such a threats becomes a key issue to detect and mitigate such attacks. For that purpose, intrusion detection systems are widely deployed network security tools aiming to detect security threats and attacks where preventive security measures are infeasible to implement [4,53].

2.2 Intrusion Detection Systems

The concept of intrusion can be defined as the set of actions or activities that compromise either one or more components of the CIA triad, the IT security model that refers to the confidentiality, integrity and availability elements of a specific system or entity. Whereas system refers not only to computers, firewall, network equipment, routers or networks but to any information technology system under the scope of the monitoring capabilities of an intrusion detection system [53]. Within this context, an intrusion detection system is a security tool which aims to detect and identify unauthorized accesses that target to misuse the system but also authorized accesses which abuse of their privileges within the system [53]. Four main approaches are used to build intrusion detection systems: misuse, anomaly, specification and hybrid [4,11,53,58]. They are outlined as follows:

- Misuse detection systems use known fingerprints or signatures of attacks stored in a database. The IDS tries to find a match between the known signatures and the current activities within the system. If a match is found, the alarm is raised about the detected suspicious behavior. Also known as signature-based systems, they are prone to be easily bypassed by unknown and novel attacks, where the signature is not yet available.
- Anomaly-based detection systems are dependant on the creation of a typical or normal activity profile or pattern. Current actions within the system are compared against the normality pattern. If the IDS finds a significant deviation or discrepancy from the normality model, the alarm is raised about the suspicious behavior. These systems are capable of detect novel attacks but they are prone to false alarms or false

positives (i.e., normal behavior is detected as malicious behavior) as the normality pattern might be difficult to model accurately. Thus, these systems are sensitive to the correctness of the normality model created.
- Specification-based detection systems use a combination of features of misuse and anomaly approaches. More concretely, anomaly-based principles are applied on a set of human-generated specifications or constraints about the normal or legitimate behavior. These systems aim to detect novel attacks using the anomaly principles and improve the limitations of anomaly-based models by reducing the amount of false positives.
- Hybrid detection systems combine any of the previous approaches, with the purpose of overcoming the weaknesses of a particular approach with the strengths of another.

The anomaly-based approach is one of the most used and effective detection methods, enabling to detect novel attacks with the inevitable trade-off of being sensitive to the correctness of the generated normality model. In this regard, statistical methods and machine learning algorithms are usually used to build the normality profile [58]. Therefore, valid behavioral models must be used to optimize the benefits obtained when using this approach, which is directly dependant on the training data used [8]. In the specific case of an IoT network, where a wide variety of devices may coexist, it is highly likely to have different normality profiles. This fact evidences the actual need of accurate IoT behavioral data which enable the implementation of effective anomaly-based intrusion detection systems. However, there is a significant lack of available data addressing the different network architectures, devices and behaviors that can be found in IoT networks and its major threats. As a result, in order to build intrusion detection systems for effective intrusion detection in IoT networks the use of proper IoT behavioral data is key.

2.3 Literature Review on Machine Learning-Based IDS

The application of machine learning to computer botnet detection first and lately to the specific case of IoT botnet detection has demonstrated encouraging results [31,58]. The noteworthy increase in IoT-related security incidents has provoked the reorientation of researchers' focus to the IoT field, thus promoting the investigation of effective and feasible IoT botnet detection methods involving anomaly-based machine learning approaches. The main aim of these approaches is to overcome the intrinsic hardware and software constraints and limited capabilities of these devices related to security [58].

In [36], Deep Autoencoders, Local Outlier Factor, One-Class Support Vector Machines and Isolation Forest models were built and tested using the N-baiot dataset. All models, except Isolation Forest, effectively detected all the simulated attacks using Mirai and BashLite malware. Deep Autoencoders provided the lowest ratio of false positives and provided the fastest attack detection times. Logistic Regression algorithm was used in [44] to estimate the likelihood for a device to be part of an IoT botnet by analyzing the connection initiation at the propagation stage. [30] developed an IoT botnet detection method combining Artificial Fish Swarm and Support Vector Machines

algorithms. A different method was proposed in [48] using Convolutional Neural Networks and binary visualization technique feed with network traffic. The novel detection approach provided fast detection times for zero-day malware. A novel application for a text recognition deep learning algorithm (Bidirectional Long Short Term Memory based Recurrent Neural Network) was developed in [33]. The suggested approach demonstrated remarkable success on Mirai botnet attack detection. In [15], different network features were used to build and assess the accuracy of traditional machine learning algorithms such as k-Nearest Neighbors, Support Vector Machines, Decision Tree, Random Forest and Artificial Neural Networks to test their detection capabilities on Mirai DDoS attacks. Traditional unsupervised anomaly-based learning algorithms were used on [39] to perform botnet detection using reduced feature sets by applying different feature selection techniques. Dimensionality reduction and discriminatory feature analysis was performed in [3] to build fast, efficient and interpretable models. Hybrid feature selection methods were evaluated in [19] to induce faster and more efficient IoT botnet detection methods.

As can be noted, the implementation of anomaly detection requires the acquisition of malicious data that is tested against the normality patterns in order to assess the goodness of the proposed detection model. Therefore, the used data sets must provide both kinds of network traffic data in order to assess effectively the detection of the threats. In this regard, we provide demonstrability of the generated data set both on classification issues (i.e., supervised learning), for the easiness of interpretation of the results and comparison and also on anomaly-based scenarios (i.e., unsupervised learning). This facts evidences the suitability of this data set to build effective anomaly detection models.

2.4 IoT Botnet Attack Anomaly Detection Datasets

As already mentioned, an vast amount of scientific literature deals with the botnet detection phenomenon in computer networks [16,17], with many publicly available data sets for experimentation [47]. On the contrary, the more recent IoT botnet phenomenon has not attracted the required attention yet, evidenced by a notable scarcity on available data sources. Table 1 provides and overview of the publicly released IoT botnet data sets which are used to build and test IoT anomaly-based intrusion detection systems. As can be observed, a small amount of data sets are available, showing common and similar characteristics in their scenarios.

Mirai, the most prominent IoT botnet and perpetrator of record-breaking attacks [9], is deployed in the vast majority of the data sets (all except Bot-IoT). Mirai, "the future" in japanese, discovered in 2016, was designed to exploit vulnerabilities on low-secured Linux-based IoT devices (i.e., consumer devices such as cameras, printers and routers) to perform massive DDoS attacks [1,9,57]. Mirai is capable to perform 10 different DDoS attacks, which can be customized using several parameters [57]. Since the release of the source code, it has been used as a basis to create other IoT botnet malware [13], but also facilitated its deployment it in a contained manner in lab environments, improving knowledge and data set creation. BashLite, also known as Lizkebab, Qbot, Torlus, Gafgyt and LizardStresser, is a notorius botnet in the IoT botnet landscape and the second most deployed in the data sets. Discovered in 2014 and made public in 2015,

it is the antecessor of Mirai [45]. As one of the oldest IoT malware, there are many variants of this malware in the wild. Since its inception, BashLite was designed to exploit devices running BusyBox (e.g., routers) evolving later to exploit any IoT device, thus enhancing its possibilities to be perform large-scale DDoS attacks [54].

Half of the data sets use emulated IoT devices, usually running on a *Raspberry Pi*. Emulators are a cheap and more scalable alternative to the usage of real IoT devices in lab environments, thus preferred in some cases. The IoT landscape embraces a wide variety of different devices used for different applications, so that is also the case for the type of devices used on the data sets, showing a great variability among data sets. Camera is the only IoT device type that can be found in more than 2 data sets. Regarding the data format, all the studies except one provide the raw *pcap* file while some also provide an structured feature data set. N-Baiot data set provides only structured data thus restricting the possibilities of perform further experimentation using this data set.

When analyzing the data sets based on the botnet lifetime cycle they encompass [22], it is shown that none of the data sets encompass all the botnet life-cycle steps, thus focusing the majority of them on the attack and post-attack phases. The data sets simulate different attacks that botnets can perform and also the scanning attack for the recruitment of new members, part of the post-attack stage. MedBIoT data set is the only data set that deals with the early stages of botnet deployment, focusing on formation and C&C stages, two of the core components of botnet deployment [29]. In this sense, this data set provides the opportunity to perform early detection of the threat, previous to the perpetration of an attack, key to prevent attacks and botnet growth.

3 Method

The main contribution of this study is the generation of a fully-labelled behavioral IoT data set and the demonstration of its suitability to induce effective machine learning-based detection systems. The data set is composed of normal and actual botnet malicious network data acquired in a medium-sized IoT network infrastructure (i.e., 83 IoT devices). The focus was placed on the acquisition of network traffic from all the endpoints and servers during the initial propagation steps performed by Mirai, BashLite and Torii botnet malware.

3.1 IoT Network Topology

The network infrastructure topology built for the purpose of this research is provided in Fig. 1. As can be observed, it is composed of three connected networks: internet network, monitoring network and IoT LAN network. Their roles, tasks and components are described as follows:

- The internet network is directly connected to the Internet and provides internet connectivity to the whole setup, for the initial configuration of different devices. A different sub-network mask is on place to restrict the connectivity between networks.
- The monitoring network provides the storage and processing capabilities for the data set creation. It receives the network data from the switch. It is composed of:

Table 1. Data sets for IoT anomaly-based IDS.

Name	Botnet	Number of devices	Device type	**R**eal or **E**mulated	Net. Size	Data format	Date	References
N-Baiot	Mirai BashLite	9	Doorbell Webcam Thermostat Baby monitor Security Camera	R	Small	structured	2018	[35,36]
IoT host-based datasets for ID research	Hajime Aidra BashLite Mirai Doflo Tsunami Wroba	2	Multimedia Center Security Camera	E	Small	pcap structured	2018	[6,7]
IoT Network Intrusion Dataset	Mirai	2	Speaker Wi-Fi Camera	R	Small	pcap	2019	[24]
Bot-IoT	No actual malware - simulated	5	Refrigerator Smart Garage door Weather Monitoring Smart Lights Smart thermostat	E	Small	pcap structured	2019	[26,38]
Aposemat IoT-23	Mirai Torii Trojan BashLite Kenjiro Okiru Hakai IRCBot Muhstik Hide&Seek	4	Raspberry Pi Lamp Amazon Echo Lock	R	Small	pcap	2020	[41]
MedBIoT	Mirai BashLite Torii	83	Switch Fan Light bulb Lock	E+R	Medium	pcap structured	2020	[20,21]

  - Capture server: responsible for the collection and storage of all the network packets captured within the whole network infrastructure. In our setup, *Tcpdump* was used to monitor and log the network traffic. Data was stored as *pcap* file format, which was later further processed by the SIEM server.
  - Security Information and Event Management (SIEM) server: responsible for data indexing, filtering, analysis and data set generation (i.e., data processing and labelling). In our setup, the SIEM server was a *Splunk* software instance.
- The IoT LAN network is a local area network (LAN) which allows malware spreading in a contained manner. It is composed of both real and virtual IoT devices. These devices generate all the behavioral traffic (i.e., benign and malicious) that is collected by the monitoring network. Containerization software (i.e., *Docker*) was used to

deploy the virtual devices. The composition and capabilities of the network devices are outlined as follows:

- Router: this networking device is responsible for the creation of an isolated network segment thus allowing only communication internally between devices. This is achieved using firewall rules. It also assigns IP addresses to the internal devices using Dynamic Host Configuration Protocol (DHCP).
- Switch: this networking device is responsible for the acquisition and transfer of the network packets. This is achieved using *port mirroring* technique. *Port mirroring* is used to clone and transfer network packets flowing through a port to another port. It can be made in real time and without any affectation on the network's performance. In this setup, all the data generated by all devices was captured and transferred to the monitoring network by this mean.
- IoT Management System: this software allows the management of all the IoT devices within the network in a single and centralized point. In our setup, it was deployed using *Hassio* software on a *Raspberry Pi*, allowing to simulate the same network behavior of the real implementations. Four types of IoT devices were emulated: switch, light bulb, lock and fan. Each type allows the remote management of different features. For instance, the emulated fan allows to turn on/off, speed selection, oscillation state and get current fan state.
- Virtual IoT devices: these IoT devices were virtualized using *Docker* containers. They are deployed using a *Raspberry Pi* thus emulating the behavior of an IoT device.
- Wireless Access Point: this networking device provided network connection to the non-ethernet compatible devices. To avoid the existence of duplicated IP addresses, it is configured to delegate on the router the capabilities of assigning IP addresses (via DHCP).
- BashLite C&C server: this server acts as the command and control unit for the BashLite malware botnet. In order to allow the spreading of the malware within the network, web and FTP services are installed. It also performs the compilation of the malware binaries used to propagate the malware infection.
- Mirai C&C server: this server acts as the command and control unit for the Mirai malware botnet. Role and tasks are analogous to BashLite C&C server.
- DNS server sinkhole: this server has two main tasks. It provides domain name resolution for the Mirai botnet and it acts as a sinkhole for the connection requests to the domains that Torii malware performs. This task provides effective malware contention within the network by avoiding the actual connection between Torii and the domain of its remote C&C server.
- Physical devices: these are the real IoT devices deployed within the network. Three different devices were used: Sonoff tasmota smart switch, TpLink smart switch and TpLink smart bulb. All these devices allow external device management and control of different features. For instance, the light bulb provides control of turn on/off, light intensity and get the device status.

To create a medium-sized network, 80 virtual devices are created and 3 physical devices are deployed. As a result, the total amount of IoT devices deployed in the LAN network is 83. The virtual devices have ARM architecture, inherited from the *Raspberry*

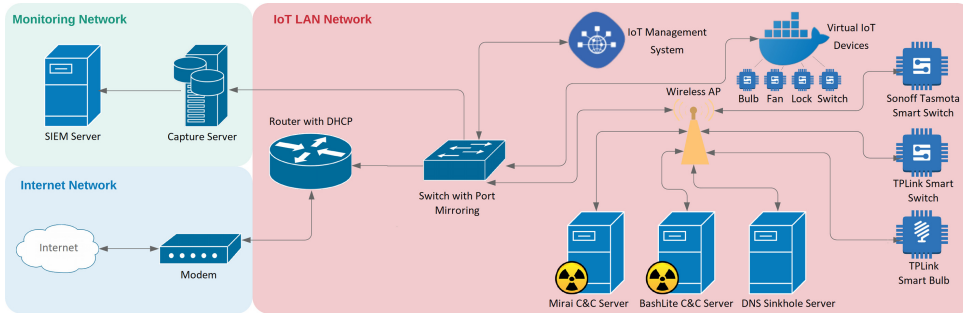


Fig. 1. Medium-sized IoT network topology. Extracted from the original paper [21].

Pi used to create them. All the real devices have MIPS architecture. The architecture of the device determines the malware binary used to infect the device. Having different types of architectures generates a wider variety of devices which enriches the spectrum of the data collected.

Regarding the virtual devices deployed, 20 instances of each type were created (i.e., fan, switch, lock and light bulb). All these devices provided different features to be controlled remotely. More specifically, all the devices allowed to active/deactivate them (on/off status). Additionally, the fan allowed to control its speed and oscillation and the light bulb its intensity.

Regarding the real devices, one instance of each type was in place (i.e., two different switches and a light bulb). All the devices allowed to active/deactivate them remotely. Additionally, the light bulb allowed control of intensity.

3.2 IoT Behavior

To simulate the behavior of IoT devices different approaches can be used, ranging from the manual usage of the devices aiming to mimic the behavior to a more automated solution using *scripts* to trigger scheduled functions/tasks. The quality and consistency of the simulated behavior is the most important element on the generation of a high quality data set that could be used as a realistic input data on effective intrusion detection systems. In such cases, the collection of relevant and real statistics of normal usage patterns offers a realistic baseline for the behavior simulation. As an example, in a average living room, a light bulb has a average usage of 1.7 h per day while in a kitchen it reaches 2.3 h [18]. These statistics provided the baseline for the simulation of normal behavior in our experimental setting. In the case of malware, the behavior was simulated by the execution of the different modules within the botnet, providing a real output of the actual botnet malware behavior.

Normal Behavior. An automated approach is selected for the simulation of the benign or normal behavior. It takes into account the architecture of the device and its performed using a Python *script* and MQ Telemetry Transport (MQTT) protocol. MQTT is a communication protocol used to manage IoT devices. The IoT management system

provides the capabilities to automate and perform scheduled tasks on the controlled IoT devices. The *script* contained the trigger actions to be performed, conveyed to the endpoints using MQTT protocol. The following triggers were used in this research setting to simulate the legitimate behavior:

- All devices are activated at 8.00 AM
- Each time a device state changes, the management system starts a countdown for the next state change. The countdown value is randomized.
- The maximum limit of changes is established in 20 and a maximum of 3 h on active state is set.
- All devices are deactivated at 07.00 PM
- To simulate a working environment, the execution of the triggers is limited to weekdays

These triggers provoked the generation of network packets along the network, which are captured and stored. The acquired network packets provide the following communication data: time, protocol used, TCP stream, TCP stream size, source IP, destination IP, MAC addresses, TCP raw message and response code.

Malicious Behavior. Three prominent botnet malware are deployed within the controlled environment. Mirai [1], BashLite [32] and Torii [28] actual malware are used to generate the malicious behavior. Mirai and BashLite botnets have been widely researched and their source code is publicly available. For that reason, their deployment is fully controlled within the experimental setup using a Command & Control server for each botnet and modifying the source code to connect only with each specific C&C server. On the contrary, Torii source code is not available, thus actual samples were used to deploy it. The samples were obtained from Hybrid Analysis archive [12]. To safely contain Torii malware within the network and avoid the connection with its real C&C server, extra contention measures are in place. As a result, Mirai, BashLite and Torii malware propagation is performed and controlled in our restricted network setting using different strategies, they are summarized in the following paragraphs.

- **Botnet Malware Propagation Techniques.** Three botnet malware are deployed within the controlled environment. Mirai and BashLite source code was publicly released, thus facilitating their contention in a similar fashion. Torii needed special measures for its unknown spreading and behavior patterns.
  - Mirai and *Yakuza* version of BashLite are configured and executed after modifying the malware source code to connect with the corresponding C&C servers within the controlled network. Mirai and BashLite use dropper as a method to download and install the appropriate infection binaries in the targets, according to their architecture. Once the binary is executed, the bot daemon will run and the compromised device will become a bot.
  - Torii behavior has not been so deeply studied yet so its deployment involves further risks. To contain and mitigate the risk of improper use of the infected devices by the actual botmaster, firewall rules and a DNS sinkhole are used. Thus Torii connection attempts with the remote C&C server are permanently

denied and redirected to the sinkhole. As a result of the actual lack of knowledge about Torii spreading methods and information about its source code, the infection binary is manually deployed in the infected devices. The sample used is tailored to infect ARM devices. The malware is run by executing the binary as root in the target devices, allowing to spread the malware through the IoT devices.

– **Botnet Contention Methods.** One of the greatest risks of deploying actual malware is the abuse of the infected devices by the real attackers. In the case of Torii, its unknown spreading methods and lack of knowledge about this malware this implies a greater challenge. In the eventuality of an unsuccessful malware contention, real attackers might be able to control the infected devices and use them to perpetrate attacks or collect relevant data. In this regard, two major risks are identified and addressed:
  1. The possibility of hidden code in Mirai's source code to establish connection with the actual C&C server
  2. Lack of knowledge about Torii's spreading techniques and capabilities

  Although Mirai spreading method is well-known, extra security measures are taken to contain the malware effectively. Firewall rules and a DNS sinkhole are in place to avoid any effective connection to the real C&C servers. The sinkhole purpose is to redirect the connection attempts, resolving the name resolution request with a controlled IP address. The firewall rules placed on the router allow to control and block traffic based on known network masks.

In this experimental setup, the three botnet malware were deployed at different times within 6 days (i.e., each let run free for 2 consecutive days). The main aim was to obtain relevant botnet data while eliminating the risk of data overlapping between different malware. Additionally, one of the Mirai malware capabilities is to detect other malware running on devices and remove it, to take the single control of it. A limited and randomized number of devices are infected for each botnet deployment. Thus, 40 devices were infected using BashLite malware, selected in a pseudo-randomized way by constraining the scope of the reachable devices. 25 devices were infected by Mirai malware, limited by restricting the internal scanner to spread on the network IP ranges. Finally, Torii malware was manually deployed in 12 devices, all under the scope of the DNS sinkhole.

3.3 IoT Behavior Verification

The generated data set was further processed and machine learning models were induced. The purpose of this experimental implementation is to verify the suitability of the generated data set for machine learning-based intrusion detection systems. In this regard, classification and anomaly detection machine learning models were built. They are briefly described as follows:

– Classification models are a kind of supervised learning which main aim is to correctly predict the class or label of an unknown data point based on specific features, also called predictors, found on the training data provided during the model building phase. When the data points are labelled into two mutually exclusive classes or

categories (e.g., legitimate and malicious traffic), binary classification is used, while when more than two categories are present in the data (e.g., legitimate, Mirai, BashLite and Torii), multiclass or multinomial classification is performed. In order to validate the outcome of this research, both approaches are implemented and validated using k-fold cross validation.

– Anomaly detection models are a type of unsupervised learning that aim to detect and identify observations that do not conform with an expected pattern or feature values in a data set. Commonly, legitimate data are used to build models that aim to detect and identify observations (i.e., malware generated points) that deviate significantly from the learnt expected normality pattern. Eventually, any data point that deviates significantly, thus seemingly not belonging to the same (legitimate) data distribution, is detected and categorized them as anomalous data points or anomalies by the learning models.

Data features are extracted from the *pcap* files acquired in this experimental setup. These features are used as predictors/inputs for all the machine learning models. More specifically, the features used in this research are computed as in [37]. Thus, 100 statistical features are generated from the network traffic, encompassing different time windows. Table 2 provides an overview of the extracted features. As it is shown, statistical features are generated for 4 main categories and 5 time windows for each one.

Table 2. Feature categories.

Category	Statistic	Time window	Features
Host-MAC&IP	Packet count Mean	100 ms 500 ms	15
Network Jitter	Variance	1.5 s	15
Channel	Packet count Mean Variance Magnitude Radius Covariance	10 s 1 min	35
Socket	Correlation		35

After the extraction of the features, a random sample for each class is generated and used to train and test 10-fold cross validated machine learning models. Four traditional machine learning classification algorithms are implemented: k-Nearest Neighbors (k-NN), Support Vector Machines (SVM), Decision Tree (DT) and Random Forest (RF). The main objective of these classifier models is to demonstrate the suitability of the generated data set for machine learning-based anomaly detection and classification models. No hyper-parameter optimization was performed, leaving room for improvement on the induced models. In this regard, default *scikit_learn* library (version 0.22.2) values are used. For each model, four performance metrics are reported: accuracy, precision, recall and F_1 score. They are defined as follows:

– Accuracy: ratio of the correctly classified test instances from all test instances.
– Precision: fraction of positive instances correctly classified among all the positive classified instances.
– Recall: fraction of positive instances correctly classified among all the actual positive instances.
– F_1 score: harmonic mean of precision and recall.

All the metrics reported range from 0 to 1. In this sense, a reported value closer to 1 is generally deemed as a positive or good result for the given metric while a value close to 0 as a bad or negative performance. Thus, for classification tasks, the greater the value the better the classifier's performance on label discrimination task, thus evidencing that the data and the classifier are suitable for that purpose. In our case, obtaining values closer to 1 in all classifiers' metrics could be used to infer that the data is suitable for machine learning-based IoT botnet detection and also that the data labels (e.g., legitimate and malware) can be effectively discriminated.

4 Results

4.1 IoT Botnet Data Set

All the network packets generated within the IoT LAN network were collected and redirected using port mirroring to the monitoring network. There, a SIEM software instance was used to perform data processing and labelling, creating the final data set. The data set is generated in two formats: structured (i.e., tabular features are extracted from the raw data) and unstructured (i.e., raw *pcap* files). The number of packets captured and provided within the whole data set are provided in Table 3.

Table 3. Data set composition.

Data source	Number of devices	Number of packets	Proportion
Normal	83	12,540,478	70.27%
BashLite	40	4,143,276	23.22%
Mirai	25	842,674	4.72%
Torii	12	319,139	1.79%
All	All	17,845,567	100%

As can be observed, the majority of the traffic is deemed as normal or legitimate IoT traffic (i.e., 70.27%) while around 30% is originated and acquired from different IoT botnet malware sources. The SIEM software used (i.e., *Splunk*) allowed further analysis and acquisition of more fine-grained of the communication details. In this regard, 32% of the normal network traffic is related to system updates, 53% to device communication (i.e., MQTT protocol) and 15% to other network data (e.g., TLS errors, pings, etc.). Regarding the malicious data, 68% of the traffic is related to malware propagation

actions while 32% to direct communication between bots and C&C servers. It is worth to note that Mirai and BashLite source codes were configured to perform these different types of communications using different ports, thus facilitating the posterior data analysis. Torii's data only includes traffic regarding the initial infection of the devices as the contention measures were explicitly established to avoid any remote communication with the actual C&C, thus preventing posterior botnet events such as propagation. The generated data set is made publicly available in the following url: https://cs.taltech.ee/research/data/medbiot/.

4.2 IoT Behavior Verification

Binary Classification. Four traditional and widely used machine learning classification models for binary classification (i.e., two class classification) are induced and 10-fold cross validated. To perform such a classification task, the data is split into two groups or labels: legitimate/normal and malware. Thus the malware class contains mixed data from the three malware deployed. More specifically, the legitimate class data is composed of 15000 randomly selected data points from the acquired legitimate traffic. The malware class is composed of 5000 randomly selected data points for each of the malware deployed within the network, summing up to 15000 data points for this class. As a result, a balanced data set is generated and used for the binary classification task. The results of models built are provided in Table 4. The table does not reflect the performance metrics for Support Vector Machines algorithm as it showed poor performance in all the assessed metrics.

Table 4. Binary classification.

Model	Acc.	Prec.	Rec.	F_1
k-NN	0.8871	0.9034	0.8871	0.8842
DT	0.9541	0.9582	0.9541	0.9538
RF	0.9702	0.9731	0.9702	0.9700

Table 5. RF confusion matrix.

		Predicted	
		Malware	Legitimate
Actual	Malware	1443	57
	Legitimate	19	1481

As can be observed, Random Forest model is able to discriminate effectively the vast majority of the network traffic, as over 97% of the data points are detected correctly. k-NN and Decision Tree models reported lower discriminatory capabilities but with performance metrics over 88% and 95%, respectively. The confusion matrix provided in Table 5, extracted from a Random Forest model, confirms that the mixed malware traffic is effectively discriminated from the normal traffic with a few misclassified points. As already stated, SVM results are not reported as they showed poor performance on all metrics. This fact may suggest that the data is not linearly separable, thus linear classifiers such as SVM or Logistic Regression may not be suitable for the classification task using this data set. Nevertheless, the results obtained using the other algorithms evidence the effective capabilities of machine learning approaches to detect botnet malware traffic, in the first stages (i.e., infection, propagation and communication with the C&C server stages) and disregarding the malware type. Furthermore, it is demonstrated that the data set generated in this research is suitable to be used as a

medium-sized realistic IoT data set for IoT botnet detection scenarios and IDS training and testing purposes.

Multiclass Classification. For this task, the data set was divided in four classes or labels according to the data source: normal, Mirai, BashLite and Torii. Four-class or multiclass classification models were induced and 10-fold cross validated using the same algorithms as in the binary task. The data set used was generated by random selection of 10000 data points for each of the classes, summing up to 40000 data points. The data set is balanced, thus data points were evenly distributed within the four labels. The purpose of this task is not only to test discrimination capabilities of legitimate/malware labels but also the discrimination of the specific malware source. Table 6 shows the results obtained for this task. As in the binary approach, Support Vector Machines algorithm is not reported, showing a poor performance in all metrics.

Table 6. Multiclass classification.

Model	Acc.	Prec.	Rec.	F_1
k-NN	0.8990	0.9073	0.8990	0.8958
DT	0.9379	0.9478	0.9379	0.9347
RF	0.9617	0.9692	0.9617	0.9602

Table 7. RF confusion matrix.

		Predicted			
		Mirai	BashLite	Torii	Leg.
Actual	Mirai	983	3	3	11
	BashLite	14	974	2	10
	Torii	5	5	978	12
	Leg.	11	3	3	983

As can be observed, in a similar fashion as in the binary models, Random Forest model outperforms Decision Tree and k-NN algorithms in the multiclass classification task. More specifically, RF algorithm provides similar discrimination performance in the multiclass task and in the binary setting, achieving over 96% accuracy in all metrics. The Random Forest model confusion matrix provided in Table 7, emphasizes the significant accuracy of this classification model in all cases, not being significantly biased towards any of the possible classes. These results suggest that network traffic source can be effectively discriminated, even in the earliest stages of botnet infection. It also demonstrates that the learning capabilities of machine learning-based detection methods can be accurate both in the general detection task (i.e., legitimate vs. malware) and in the detection of different sources of malicious traffic in medium-sized IoT networks.

Anomaly Detection. This task involves the identification of abnormal or unusual observations within the data distribution, the so-called *anomalies*. In our case, these abnormal observations are the ones generated by the non-normal behavior of the IoT devices, mainly caused by malware activity. Depending on the purpose of anomaly detection it can be further divided into *outlier* detection and *novelty* detection. In outlier detection, the training data set contains outliers (i.e., observations that deviate significantly from the rest) and the goal of the algorithms is to identify regions where data is most concentrated thus ignoring the data that lie far from that regions, the *outliers*. The observations on concentrated areas, which belong to the same data distribution,

are called *inliers*. In the case of novelty detection it is assumed that the training data set does not contain outliers and the goal is, given a new observation, detect whether it can be categorized as an outlier or an inlier. In this case, the outlier observation is called *novelty*. Both anomaly detection approaches were used and tested with the data set generated in this research. The anomaly detection algorithm used was Local Outlier Factor (LOF), which is capable to perform both novelty and outlier detection tasks. LOF's algorithm *Scikit-learn* library implementation was used to build and test all the anomaly-based models [46].

In order to build the models, the data was sampled using random sampling. Thus, for each benign data set 100.000 observations were randomly selected (i.e., 90.000 for training and 10.000 for testing) and 10.000 observations for each malware data set (i.e., BashLite, Mirai and Torii). Prior to induce the models, data was pre-processed by using standardization and Principal Component Analysis (PCA) to reduce data dimensionality, from high dimensionality data to lower dimensionality (i.e., ranging from 10 to 30 Principal Components on all induced models). Principal components are new generated features by PCA algorithm, created from the linear combination of the original features of the data set, aiming to capture the maximum variance within the data points. Based on that, the new generated features possess no real meaning or category and they are just called Principal Components or PC (i.e., 1PC is the first principal component). Two different scenarios were tested in the anomaly-based induced models. In the first scenario, legitimate data captured during the time a specific malware was running was used to build the models. The testing sets correspond to held-out legitimate data and malware data from that specific collection time-frame. For example, as can be observed in the first row in Table 8, the training data corresponds to legitimate data acquired during the deployment of BashLite malware. The testing samples correspond to legitimate data from the same period of time and BashLite malware generated data. The detection performances for this first scenario are provided in Table 8. The column *training*, specifies the normal data training source used to build the corresponding model while the *test malware* and *test normal* refer to the source of data used for testing purposes. The *mixed total* column provides the average of the previous two columns, as the same amount of legitimate and malware instances were tested against the model (i.e., 10.000 samples). The *All* value refers to a stratified mix of the legitimate data (i.e., 1/3 of each of the previous data sets). The performance metric used is accuracy, which provides the ratio of correctly classified instances among all the testing samples. Accuracy ratios closer to 1 indicate a good performance metric while closer to 0 a poor detection performance.

Table 8. Novelty detection performance - first scenario.

Training	Test normal	Test malware	Mixed total
BashLite	0.9486	0.9628	0.9557
Mirai	0.9331	0.8552	0.8942
Torii	0.9433	0.9515	0.9474
All	0.9444	0.9129	0.9286

As can be observed on Table 8, models built with BashLite, Torii and combined legitimate data provide detection performances over 91% on malware and over 93% on the held-out legitimate data. Legitimate data belonging to Mirai deployment provides less accuracy on the malware data and test data, suggesting that the malware is more similar to legitimate traffic but prone to be discriminated effectively. It is worth to note that these models are not optimized and there is room to improvement as mostly default values were used on the generation of the models. According to the results, BashLite malware provides a differentiated profile from normal traffic that make the models more effective in the detection of this specific malware. Torii and the mixed model (i.e., using stratified randomly sampled legitimate data from the three data sets) provide high accuracy ratios for malware detection based on anomaly models. In any case, these results evidence IoT malware can be discriminated from legitimate and effectively detected using anomaly-based detection models in the early stages of a botnet deployment (i.e., prior to any attack).

In the second scenario, the same models built on the first scenario were tested against other test sets belonging to different malware data. For example, the first row in Table 9, the training data corresponds to legitimate data acquired during the deployment of BashLite malware. The testing samples correspond to the same time-frame BashLite generated data, and also data belonging to the deployments of Torii and Mirai malware. This setting allows to test the goodness of the anomaly detection models to detect different types of malware. The column *training* in Table 9 specifies the normal data training source used to build the corresponding model while the rest of columns specify what malware data test was tested. The *All* value refers to a stratified mix of the legitimate data (i.e., 1/3 of each of the previous data sets). The *Test Mixed* column provides the performance when a mixed data set of the three malware data sets was combined and tested against the model. This test data set was generated using stratified random sampling (i.e., the same amount of samples extracted from each malware data set, 33%). The performance metric reported is the detection accuracy.

Table 9. Novelty detection performance - second scenario.

Training	Test Mirai	Test Torii	Test BashLite	Test mixed
BashLite	0.9066	0.9842	0.9628	0.9536
Mirai	0.8552	0.9665	0.9643	0.9262
Torii	0.8839	0.9515	0.9618	0.9120
All	0.8407	0.9594	0.9615	0.9074

The results provided in Table 9 suggest that the anomaly-based detection models built in the first scenario are capable to detect effectively not only its specific malware but also the other IoT botnet malware. With the exception of the detection of Mirai malware, which is slightly worse than the other malware, the detection ratios are over

91% in all models, whatever the data source used, except the Mirai-based model. These results emphasize the goodness of the anomaly-based models to detect malware effectively and the goodness of the generated data set to build effective anomaly-based IoT malware detection models on early stages of botnet deployment.

5 Conclusions

The Internet of Things is growing exponentially and these devices will become ubiquitous in the following years. This fact, in combination with the traditional lack of security measures associated to them, make IoT devices an appealing objective for cyber attackers. The vulnerable IoT devices are compromised and become part of a *botnet*, which is mainly used as amplification platform for cyber attacks. In this regard, botnets have been used to perpetrate massive DDoS attacks against companies and individuals, leading to nefarious consequences. As a result, the security of these devices is a critical issue to be addressed. The most recent solutions involve machine learning techniques, which are providing notable and promising results.

The performance of machine learning models is directly related with the amount of data used to build the models and its quality to capture the phenomenon. In the specific case of IoT botnet detection, there is a remarkable lack of data sets which limits the possibilities of building efficient machine learning-based models. This paper elaborates on the original research where MedBIoT data set was introduced [21] by adding more experimentation with the acquired data, demonstrating the suitability of MedBIoT data set to build effective machine learning-based IoT botnet detection models. As provided in [21], the data set focuses on the early stages of botnet deployment in a medium-sized IoT network (i.e., 83 IoT devices). Three prominent botnet malware were deployed (i.e., Torii, Mirai and BashLite) in different IoT devices within the network. The network traffic data is provided labelled according to its source: botnet malware or normal.

Supervised (i.e., classification) and unsupervised (i.e., anomaly detection) machine learning models are induced and tested. The obtained results evidence the goodness of MedBIoT data set to build effective IoT botnet detection models, using both machine learning-based approaches. In this regard, the performance metrics obtained in all the tested scenarios (i.e., over 85% in all cases) prove that IoT botnet detection can be achieved with high accuracy even in the early stages of botnet deployment, thus preventing the attack phase and avoiding its nefarious consequences. As a result, MedBIoT data set complements the existing data sets, which mainly focus on attack scenarios, by putting emphasis on the early stages of botnet deployment. Early detection may help to prevent attacks and botnet growth in a significant manner.

The extensive experimentation performed in this research proves the suitability of MedBIoT data set as a reliable data source for IoT botnet detection in general and intrusion detection systems' testing, design and deployment in particular. The data set is available at https://cs.taltech.ee/research/data/medbiot/.

References

1. Antonakakis, M., et al.: Understanding the mirai botnet. In: 26th $USENIX$ Security Symposium ($\{USENIX\}$ Security 17). pp. 1093–1110 (2017)
2. Asokan, A.: Massive botnet attack used more than 400,000 IoT devices (2019). https://www.bankinfosecurity.com/massive-botnet-attack-used-more-than-400000-iot-devices-a-12841
3. Bahşi, H., Nõmm, S., La Torre, F.B.: Dimensionality reduction for machine learning based IoT botnet detection. In: 2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV), pp. 1857–1862 (2018)
4. Benkhelifa, E., Welsh, T., Hamouda, W.: A critical review of practices and challenges in intrusion detection systems for IoT: toward universal and resilient systems. IEEE Commun. Surv. Tutor. **20**(4), 3496–3509 (2018)
5. Bertino, E., Islam, N.: Botnets and internet of things security. Computer **2**, 76–79 (2017)
6. Bezerra, V.H., da Costa, V.G.T., Martins, R.A., Junior, S.B., Miani, R.S., Zarpelao, B.B.: Data set (2018). http://www.uel.br/grupo-pesquisa/secmq/dataset-iot-security.html
7. Bezerra, V.H., da Costa, V.G.T., Martins, R.A., Junior, S.B., Miani, R.S., Zarpelao, B.B.: Providing IoT host-based datasets for intrusion detection research. In: Anais do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, pp. 15–28. SBC (2018)
8. Bolzoni, D.: Revisiting Anomaly-based Network Intrusion Detection Systems. University of Twente, Enschede (2009)
9. Bonderud, D.: Leaked mirai malware boosts IoT insecurity threat level (2016). https://securityintelligence.com/news/leaked-mirai-malware-boosts-iot-insecurity-threat-level/
10. Bosche, A., Crawford, D., Jackson, D., Schallehn, M., Schorling, C.: Unlocking opportunities in the internet of things (2018). https://www.bain.com/contentassets/5aa3a678438846289af59f62e62a3456/bain_brief_unlocking_opportunities_in_the_internet_of_things.pdf
11. Butun, I., Morgera, S.D., Sankar, R.: A survey of intrusion detection systems in wireless sensor networks. IEEE Commun. Surv. Tutor. **16**(1), 266–282 (2013)
12. Crowdstrike: Hybrid analysis (2019). https://www.hybrid-analysis.com/
13. DeBeck, C., Chung, J., McMillen, D.: I can't believe mirais: tracking the infamous IoT malware (2019). https://securityintelligence.com/posts/i-cant-believe-mirais-tracking-the-infamous-iot-malware-2/
14. Doffman, Z.: Cyberattacks on IoT devices surge 300% in 2019, 'measured in billions', report claims (2019). https://www.forbes.com/sites/zakdoffman/2019/09/14/dangerous-cyberattacks-on-iot-devices-up-300-in-2019-now-rampant-report-claims/#574229995892
15. Doshi, R., Apthorpe, N., Feamster, N.: Machine learning DDoS detection for consumer internet of things devices. In: 2018 IEEE Security and Privacy Workshops (SPW), pp. 29–35. IEEE (2018)
16. Feily, M., Shahrestani, A., Ramadass, S.: A survey of botnet and botnet detection. In: 2009 Third International Conference on Emerging Security Information, Systems and Technologies, pp. 268–273. IEEE (2009)
17. Garcia, S., Grill, M., Stiborek, J., Zunino, A.: An empirical comparison of botnet detection methods. Compu. Secur. **45**, 100–123 (2014)
18. Gifford, W.R., Goldberg, M.L., Tanimoto, P.M., Celnicker, D.R., Poplawski, M.E.: Residential lighting end-use consumption study: estimation framework and initial estimates (2012). https://www1.eere.energy.gov/buildings/publications/pdfs/ssl/2012_residential-lighting-study.pdf
19. Guerra-Manzanares, A., Bahsi, H., Nõmm, S.: Hybrid feature selection models for machine learning based botnet detection in IoT networks. In: 2019 International Conference on Cyberworlds (CW), pp. 324–327 (2019)

20. Guerra-Manzanares, A., Medina-Galindo, J., Bahsi, H., Nõmm, S.: Medbiot data set archive (2020). https://cs.taltech.ee/research/data/medbiot/
21. Guerra-Manzanares, A., Medina-Galindo, J., Bahsi, H., Nõmm, S.: Medbiot: generation of an IoT botnet dataset in a medium-sized IoT network. In: Proceedings of the 6th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP, pp. 207–218. INSTICC, SciTePress (2020). https://doi.org/10.5220/0009187802070218
22. Hachem, N., Mustapha, Y.B., Granadillo, G.G., Debar, H.: Botnets: lifecycle and taxonomy. In: 2011 Conference on Network and Information Systems Security, pp. 1–8. IEEE (2011)
23. Hilton, S.: DYN analysis summary of Friday October 21 attack (2016). https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/
24. Kang, H., Ahn, D.H., Lee, G.M., Yoo, J.D., Park, K.H., Kim, H.K.: IoT network intrusion dataset(2019). http://dx.doi.org/10.21227/q70p-q449
25. Kolias, C., Kambourakis, G., Stavrou, A., Voas, J.: Ddos in the IoT: Mirai and other botnets. Computer **50**(7), 80–84 (2017)
26. Koroniotis, N., Moustafa, N., Sitnikova, E., Turnbull, B.: Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-IoT dataset. Fut. Gene. Comput. Syst. **100**, 779–796 (2019)
27. Krebs, B.: Krebsonsecurity hit with record Ddos (2016). https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/
28. Kroustek, J., Iliushin, V., Shirokova, A., Neduchal, J., Hron, M.: Torii botnet - not another mirai variant (2018). https://blog.avast.com/new-torii-botnet-threat-research
29. Leonard, J., Xu, S., Sandhu, R.: A framework for understanding botnets. In: 2009 International Conference on Availability, Reliability and Security, pp. 917–922. IEEE (2009)
30. Lin, K.C., Chen, S.Y., Hung, J.C.: Botnet detection using support vector machines with artificial fish swarm algorithm. J. Appl. Math. **2014** (2014)
31. Livadas, C., Walsh, R., Lapsley, D.E., Strayer, W.T.: Using machine learning techniques to identify botnet traffic. In: LCN, pp. 967–974. Citeseer (2006)
32. Marzano, A., et al.: The evolution of bashlite and mirai IoT botnets. In: 2018 IEEE Symposium on Computers and Communications (ISCC), pp. 00813–00818. IEEE (2018)
33. McDermott, C.D., Majdani, F., Petrovski, A.V.: Botnet detection in the internet of things using deep learning approaches. In: 2018 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. IEEE (2018)
34. McKinsey: What's new with the internet of things? (2017). https://www.mckinsey.com/industries/semiconductors/our-insights/whats-new-with-the-internet-of-things
35. Meidan, Y., et al.: detection_of_iot_botnet_attacks_n_baiot data set (2018). http://archive.ics.uci.edu/ml/datasets/detection_of_IoT_botnet_attacks_N_BaIoT
36. Meidan, Y., et al.: N-baiot-network-based detection of IoT botnet attacks using deep autoencoders. IEEE Perva. Comput. **17**(3), 12–22 (2018)
37. Mirsky, Y., Doitshman, T., Elovici, Y., Shabtai, A.: Kitsune: an ensemble of autoencoders for online network intrusion detection. arXiv preprint arXiv:1802.09089 (2018)
38. Moustafa, N.: The bot-IoT dataset. http://dx.doi.org/10.21227/r7v2-x988 (2019). 10.21227/r7v2-x988
39. Nõmm, S., Bahşi, H.: Unsupervised anomaly based botnet detection in IoT networks. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 1048–1053 (2018)
40. O'Donnell, L.: More than half of IoT devices vulnerable to severe attacks (2020). https://threatpost.com/half-iot-devices-vulnerable-severe-attacks/153609/
41. Parmisano, A., Garcia, S., Erquiaga, M.J.: Stratosphere laboratory. a labeled dataset with malicious and benign IoT network traffic (2020). https://www.stratosphereips.org/datasets-iot23

42. Pratt, M.K.: Top challenges of IoT adoption in the enterprise (2019). https://internetofthingsagenda.techtarget.com/feature/Top-challenges-of-IoT-adoption-in-the-enterprise
43. Pritchard, M.: Ddos attack timeline: time to take Ddos seriously (2018). https://activereach.net/newsroom/blog/time-to-take-ddos-seriously-a-recent-timeline-of-events/
44. Prokofiev, A.O., Smirnova, Y.S., Surov, V.A.: A method to detect internet of things botnets. In: 2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), pp. 105–108. IEEE (2018)
45. Radware: A quick history of IoT botnets (2018). https://blog.radware.com/uncategorized/2018/03/history-of-iot-botnets/
46. Scikit-Learn: novelty and outlier detection (2020). https://scikit-learn.org/stable/modules/outlier_detection.html
47. Shiravi, A., Shiravi, H., Tavallaee, M., Ghorbani, A.A.: Toward developing a systematic approach to generate benchmark datasets for intrusion detection. Comput. Secur. **31**(3), 357–374 (2012)
48. Shire, R., Shiaeles, S., Bendiab, K., Ghita, B., Kolokotronis, N.: Malware squid: a novel iot malware traffic analysis framework using convolutional neural network and binary visualisation. In: Internet of Things, Smart Spaces, and Next Generation Networks and Systems, pp. 65–76. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-01168-0
49. Silva, S.S., Silva, R.M., Pinto, R.C., Salles, R.M.: Botnets: a survey. Comput. Netw. **57**(2), 378–403 (2013)
50. Sklavos, N., Zaharakis, I.D., Kameas, A., Kalapodi, A.: Security & trusted devices in the context of internet of things (IoT). In: 2017 Euromicro Conference on Digital System Design (DSD), pp. 502–509. IEEE (2017)
51. Statista: Forecast end-user spending on iot solutions worldwide from 2017 to 2025 (2019). https://www.statista.com/statistics/976313/global-iot-market-size/
52. Statista: Number of internet of things (IoT) connected devices worldwide in 2018, 2025 and 2030 (2019). https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/
53. Sun, B., Osborne, L., Xiao, Y., Guizani, S.: Intrusion detection techniques in mobile ad hoc and wireless sensor networks. IEEE Wirel. Commun. **14**(5), 56–63 (2007)
54. TrendMicro: Bashlite IoT malware updated with mining and backdoor commands, targets WeMo devices (2019)
55. Weagle, S.: Financial impact of mirai Ddos attack on DYN revealed in new data (2017). https://www.corero.com/blog/797-financial-impact-of-mirai-ddos-attack-on-dyn-revealed-in-new-data.html
56. Weisman, S.: Emerging threats - what is a distributed denial of service attack (Ddos) and what can you do about them? (2019). https://us.norton.com/internetsecurity-emerging-threats-what-is-a-ddos-attack-30sectech-by-norton.html
57. Winward, R.: IoT attack handbook: A field guide to understanding IoT attacks from the mirai botnet to its modern variants (2018). https://www.datacom.cz/userfiles/miraihandbookebook_final.pdf
58. Zarpelão, B.B., Miani, R.S., Kawakani, C.T., de Alvarenga, S.C.: A survey of intrusion detection in internet of things. J. Netw. Comput. Appl. **84**, 25–37 (2017)

Appendix 15

Publication XV
A. Guerra-Manzanares and H. Bahsi. On the application of active learning for efficient and effective early iot botnet detection. *Journal article*, *under review*, 2022

On the Application of Active Learning for Efficient and Effective IoT Botnet Detection

Alejandro Guerra-Manzanares and Hayretdin Bahsi

Department of Software Science, Tallinn University of Technology

Abstract

The active learning approach for machine learning can greatly benefit those environments where a wealth of unlabeled data is available, and the labeling cost of the data can be restrictive. In this regard, Security operating centers (SOCs) can take advantage of the human expertise available to improve machine learning-based detection models using the active learning approach. In the context of SOC operations and IoT botnet detection, our study provides a thorough benchmarking of the application of different active learning approaches within the framework of pool-based sampling. The selection of the optimal query instance for learning is evaluated using uncertainty sampling, ranked batch-mode sampling, and query by committee strategies. Our results show that the active learning approach can help to generate better detection models using all the active learning query strategies tested in our benchmarking setup. Leveraging the human-machine interaction can produce high-performance models in the context of IoT botnet detection using significantly less data than the passive approaches traditionally used for the generation of machine learning-based detection systems. Additionally, the impact of wrong-labeled data in the active learning implementation is explored.

Keywords: active learning, iot botnet, botnet detection, machine learning, query learning, internet of things, intrusion detection, iot

1. Introduction

Botnets constitute a serious threat as they are the network infrastructure utilized by the attackers who conduct large-scale malicious activities such as identity theft, denial of service attacks, or sending unsolicited messages [1]. According to Cisco, machine-to-machine connections generated by IoT devices will constitute half of the global network connections by 2023 [2]. Consequently, compromised IoT devices amplify the seriousness of this threat, as evidenced by some past incidents, such as the high-volume DDoS attacks that occurred in 2016 [3].

Security specialists resort to security monitoring systems to identify malicious activities in their networks in their early stages. Current solutions still mostly rely on signature-based detection, which requires the generation of specific attack descriptions. However, they suffer from high false-positive rates and cannot detect newly-evolved attack types prematurely. A survey conducted with IT specialists of US companies identified that only 19% of the alerts from these solutions are reliable and that only 4% are investigated by security analysts [4].

Machine learning promises to be a significant alternative solution to signature-based systems [5]. In this regard, supervised models induced for IoT botnet detection have demonstrated high performance [6]. However, finding or creating labeled data is very hard in the cyber security domain due to the confidentiality concerns of organizations or human resource shortages. Thus, unsupervised models have been suggested by the research community to overcome this problem. However, these methods still require data that are guaranteed to be *benign*. High-performance results have been obtained when a dedicated model is induced for each specific IoT device [7] . However, this is prone to cause a significant management burden on system owners and solution developers due to the enormous number of device types in the IoT landscape.

The assumption of the lack of resources to label data and the consideration of only unsupervised options underestimate the analysis capabilities of contemporary organizational structures, such as security operations centers (SOCs) or managed security service companies (MSCs), which have continuously developed during recent years. We conjecture that the effective utilization of existing but limited expert resources in such organizations could pave the way to use active learning. Active learning is an approach in which humans can assist improving the existing learning models created using a small number of labeled data. Considering the continuous incident handling cycles in SOCs and MSCs, these organizations may accumulate labeled data over time and develop well-working solutions.

In the active learning approach, which is usually considered a type of semi-supervised learning, the supervised learning algorithm selects specific samples (i.e., usually the most informative ones) from an unlabeled data set. Next, the selected instance is provided to experts for labeling. The labeled instance is used to update the knowledge of the model, which is initially induced with a small number of labeled data. The predictive capability of the learning model may depend on the quality of the instance selection process, the size of the unlabeled data pool, the number of labeled instances used to create the initial model, and the accuracy of the expert decisions.

Active learning is well-suited to problem domains in which collecting data is easy and cheap, but data labeling is expensive [8]. Intrusion detection could be one of the application domains as it is easy to collect raw network or host data from the systems and convert it to a suitable format for learning tasks.

However, assigning the relevant resources to the labeling tasks is challenging due to the limited number of human experts with sufficient security skills to perform the task. Besides, confidentiality concerns usually prevent organizations from sharing any data, be it raw or labeled, with others, exacerbating the problem.

In addition to the continuous accumulation of expert knowledge, active learning provides an instrument for balancing the required labeling effort and predictive capability, which can be optimized for an organization according to the cost and availability of experts. Therefore, the successful implementation of active learning models has the potential to guide the organizational restructuring efforts in SOCs and MSCs.

Despite the promising advantages of the active learning approach, experts may likely misclassify some instances, which may hurt the models. Therefore, it is important to consider the possible consequences of such labeling errors, which may vary depending on the experience and knowledge of experts.

In this study, we present a comprehensive benchmarking of active learning approaches for early IoT botnet detection. More specifically, we evaluated the impact of the following design components of an active learning implementation: (1) Size of the unlabeled data pool, (2) Query strategy (i.e., uncertainty sampling, query by committee, ranked batch-mode), (3) Size of the initial data set (i.e., seed size), (4) Wrong labeling. We selected a supervised model trained with a large set of instances as a reference for the comparison (i.e., passive approach). Random selection of *query* instances is also used as a baseline for evaluating the performance of the active learning query strategies. The experimental results prove that active learning constitutes an important solution in this problem domain and that can provide high detection performance even using wrong-labeled instances.

Active learning has been applied in the cyber security field in general and intrusion detection in particular in a small number of studies [9, 10, 11]. However, empirical studies in the IoT botnet detection problem domain are missing. We contemplate that most research that applies machine learning to intrusion detection focus on optimizing the detection accuracy without considering the human and machine interaction. Labeling difficulties have been acknowledged, but solution approaches have been more on resorting to benign data to induce one-class learning models. Still, obtaining benign data from operational environments or being sure that the obtained data is benign is a challenging task. However, instead of considering labeling and model development as separate tasks, both tasks can be handled holistically if the perspective of human and machine interaction is taken into consideration within the framework of active learning. The utilization of this methodology suits well the organizational developments in the cyber security field as the maturity of SOCs has developed recently.

More empirical studies in the IoT domain are needed to understand the benefits of machine learning in the intrusion detection problem. Despite the security weaknesses that characterize IoT devices, which pave the way for advanced attacks, these devices may have distinguishable benign traffic that can significantly benefit from machine learning solutions compared to usual IT networks.

This study is distinguished as it gives and discusses the results of a comprehensive benchmarking about the application of active learning to an intrusion detection problem in IoT environments. Moreover, it elaborates on the *wrong* or *noisy* labeling problem, which is highly likely to occur in real-world settings, especially in SOCs and MSCs.

This paper is structured as follows: Section 2 gives background information about the active learning methodology in general and query strategies in particular. Previous research dealing with the application of this method to cyber security problems, and specifically for intrusion detection, is reviewed in Section 3. Section 4 presents the data set used, the machine learning workflow, and the overall design of our benchmarking study. Main results are presented in Section 5 and further discussed in Section 6. Section 8 concludes the paper.

2. Background information

Given the wealth of unlabeled data available in many domains and the expensive and time-consuming labeling process, the application of active learning is well-motivated to provide high-performance systems minimizing the labeling cost. Intrusion detection is one of such domains.

Active learning is a form of *semi-supervised learning*. Semi-supervised learning systems are built by combining a small quantity of labeled data with a large amount of unlabeled data during training [12]. The core idea behind active learning is that a machine learning algorithm can provide better performance with fewer training steps (i.e., fewer data instances) if it is allowed to select the data from which it learns [13]. For such a purpose, a supervised model trained with a small amount of data (i.e., *active learner*) may pose *queries* regarding selected unlabeled data instances for labeling by an *oracle*, i.e., a human with expertise in the task. Also called *query learning*, active learning-based systems try to overcome the labeling bottleneck by letting the classifier select specific instances of interest (i.e., *query* instances), based on an *informativeness* or *relevancy* score, from a collection of unlabeled samples and asking an oracle (e.g., a human annotator) to label them [13]. This aims to remove the randomness in data labeling and focus the labeling efforts on the most relevant instances for the task, based on the selection of the classifier. By doing this, the active learning approach aims to achieve high accuracy, updating the knowledge of the model using as few labeled samples as possible, thus minimizing the cost of the data labeling process.

The selection of the specific sample for labeling (i.e., query instance) at each training iteration and updating the model capabilities is based on an *informativeness* criterion applied by the active learner using a specific query strategy to the unlabeled data set [14]. The *most informative* instance is selected for labeling, which may vary depending on the strategy, and used to retrain the model and improve its knowledge. The most common active learning approach, the pool-based framework, is depicted in Figure 1.

The present paper is a benchmarking study of pool-based strategies for active learning implementations. As shown in

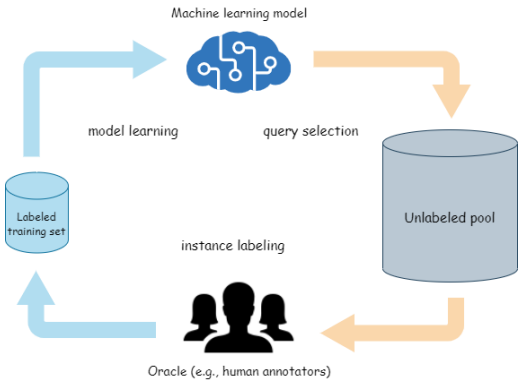


Figure 1: Pool-based active learning cycle

Figure 1, the pool-based active learning framework assumes the existence of a small set of labeled data and the availability of a large collection or *pool* of unlabeled data [13]. Query instances are selected from the unlabeled pool of samples for expert annotation (i.e., oracle). Once labeled, the sample is incorporated into the labeled training set, which is used to update the knowledge of the learning model (i.e., model retraining). For its implementation, the active learning cycle depicted in Figure 1 may be repeated until a stopping criterion (e.g., performance threshold) is met.

Query instance selection is the critical step for the success of the active learning approach. For this reason, this study evaluates different query selection strategies. The instances are usually selected greedily, based on an informativeness score used to evaluate all the samples of the unlabeled pool [14]. Thus the *most informative* sample is selected at each training step. The query strategies tested in this research, used to evaluate the degree of informativeness of the samples and select the query instance, are introduced in the following paragraphs.

2.1. Uncertainty sampling

The simplest and most commonly used query instance selection strategy. In this strategy, the query instance is selected based on how *certain* is the learner about the class label of a sample. For this purpose, different informativeness scores can be used such as:

- *Classification uncertainty (U)*: the learner selects the instance (x) for which it is least certain about how to label (i.e., the largest uncertainty score). It is based on a *least confidence* score computed as:

$$U(x) = 1 - P(y^*|x) \tag{1}$$

  where x is a specific unlabeled sample and y^* is the most likely prediction for that sample.

- *Classification margin (M)*: a score that computes the difference in probability between the first and second most likely predictions. The sample with the *smallest* margin is selected as it corresponds to the most *uncertain* decision. It is calculated as:

$$M(x) = P(y_1^*|x) - P(y_2^*|x) \tag{2}$$

  where y_1^* and y_2^* are the first and second most likely classes based on the current classification model for a specific instance x.

- *Classification entropy (H)*: this score involves the usage of *entropy* as a classification uncertainty measure. The sample with the *highest* entropy score is selected. It is computed as:

$$H(x) = -\sum_k p_k \log(p_k) \tag{3}$$

  where p_k refers to the probability of a specific unlabeled sample to belong to the kth class according to the current classifier knowledge.

2.2. Query by committee

This query strategy involves the generation of a group of classification models, called a committee, trained on the same current labeled set, but representing competing hypotheses in the hypothesis space. In this strategy, each member of the committee votes on the class label of the *query* candidates. The most informative instance is considered the one in which they most disagree about its class. Different query selection scores can be applied to evaluate the degree of disagreement, such as:

- *Vote entropy*: the query instance selected is the sample for which the entropy of the vote distribution is the largest. It is computed as:

$$VE = \arg\max_x -\sum_i \frac{V(y_i)}{C} \log(\frac{V(y_i)}{C}) \tag{4}$$

  where y_i ranges over all possible class labels, and $V(y_i)$ is the number of *votes* a label receives from the committee members for a specific instance x. C is the committee size.

- *Consensus entropy*: this disagreement measure first calculates the average of the class probabilities of each classifier, named consensus probability (P_{cs}). Then the entropy of the consensus probability is calculated. The unlabeled sample with the highest consensus entropy is selected as the query instance. The score is calculated as:

$$CE = -\sum_y P_{cs} \log(P_{cs}) \tag{5}$$

  where $P_{cs} = \frac{1}{C}\sum_{c=1}^{C} P(y_i)$ is the consensus probability.

- *Maximum disagreement*: this score also calculates the consensus probability by averaging the predicted class probabilities. Instead of entropy, the *Kullback-Leibler* divergence is used to quantify the difference between the consensus probabilities and the predicted class probabilities

for each committee member. The sample with the highest *divergence* score is chosen as query instance. The score is computed as:

$$MD = \arg\max_{x} \frac{1}{C} \sum_{i}^{C} D(P_{\theta^{(c)}} || P_{cs}) \quad (6)$$

where $D(P_{\theta^{(c)}} || P_{cs}) = \sum_i P(y_i|x;\theta^{(c)}) \log \frac{P(y_i|x;\theta^{(c)})}{P_{cs}}$ calculates the corresponding *Kullback-Leibler* divergence. $\theta^{(c)}$ represents a particular classification model of the committee, and P_{cs} is the consensus probability.

2.3. Ranked batch-mode sampling

The classic pool-based active learning framework is designed to query only one instance per retraining step (see Figure 1). The ranked batch-mode sampling strategy, proposed by Cardoso et al. [15], allows the learner to query and label multiple instances at once, aiming to overcome the limitations of the classic approach. According to this strategy, each sample of the unlabeled pool is scored using the formula:

$$score = \alpha(1 - \Phi(x, X_{labeled})) + (1 - \alpha)U(x) \quad (7)$$

where $\alpha = \frac{|X_{unlabeled}|}{|X_{unlabeled}|+|X_{labeled}|}$, $X_{labeled}$ is the labeled data set, U(x) is the *least confidence* score of a particular sample x, and Φ is a similarity function (e.g., cosine similarity). The similarity function measures how well the feature space is already explored around the sample x by the current labeled set used to induce the classification model. The samples are ranked from the greatest to the lowest score. The sample with the largest score is selected and removed from the pool. Then, the score is recalculated for the remaining instances and the process is repeated until the desired number of instances that compose the learning *batch* are selected. The *batch* or set of chosen instances is queried for labeling in a single training step and used to update the learning model.

3. Related work

Active learning focuses on selecting relevant samples around the decision boundary of the learning model to enhance the classification performance of the model using a reduced labeled training set. Despite its successful application to various problem domains [16], the active learning approach is not guaranteed to cover the whole representation space and may miss some data clusters. This is likely to cause performance degradation in some data sets (i.e., known as *sampling bias*) [8]. It is suggested that supporting uncertainty-based selection methods with random sampling may solve this problem [8].

Early implementations of active learning to the intrusion detection problem were demonstrated on the well-known *KDD Cup 1999* data set [9, 10]. In [9], a Support Vector Machine (SVM) classifier used an uncertainty approach based on the closeness of the points to the decision hyperplane and achieved a reduction by a factor of eight in the number of queries compared to the results of the random selection strategy. As a modification to the uncertainty sampling strategy, this study uses a balanced model in which the selection strategy guarantees that an equal representation of the classes is included in the training set.

Confidence measures proposed by transductive reliability estimation were adapted as query mechanism for k-Nearest Neighbors (k-NN) models in [10]. This study reached the same performance that can be achieved using 2000 randomly selected instances, with only 40 samples using their proposed active learning approach. Cost-sensitive learning methods which assign varied costs to misclassifications, instead of uniform costs, have also been adapted to the active learning context and successfully applied to the *KDD Cup 1999* dataset [17]. In [18], sampling bias is addressed using a labeling strategy that applies rare category detection in addition to the classic uncertainty strategy. This study leveraged the multi-class logistic regression algorithm to better cover the representation space by selecting representative instances from different families and demonstrated its effectiveness on two data sets: *Contagio*, a data set composed of malicious and benign *pdf* files, and *NSL-KDD*, a data set that includes network attack data. A similar idea of supporting the uncertainty strategy with anomaly detection is described in [19].

Other notable studies taking advantage of the active learning approach are [20] for detecting malicious behavior on Microsoft Office documents and [21] for phishing URLs.

Besides the detection of malicious files or network attacks, active learning strategies have also been applied to cyber threat intelligence problems. Security-related tweets obtained from leading cyber security experts were classified according to their relevancy using active learning models in [11]. In this study, uncertainty sampling was applied to label instances and update *k-NN* and *Random Forest* models using an online learning approach. The results showed that the labeling of 600 instances using the active learning approach was sufficient to reach the same performance as the model induced by random sampling using about 6000 samples.

Despite these initial and other posterior attempts, compared to the huge body of machine learning research in this problem domain [5], it can be argued that active learning has not drawn significant attention within the cyber security research community. The application of the active learning approach to the intrusion detection problem requires more elaboration for IoT networks as the characteristics of these networks show significant differences from conventional IT systems. For instance, benign network traffic between IoT devices may show some regular communication patterns that can be beneficial for the learning model to discriminate the malicious behavior successfully. On the other side, the security of IoT devices is usually neglected due to poor secure development and administration practices.

In Table 1, we compared our study with other research studies that specifically address network intrusion detection. The comparison covers the aspects such as baselines, sampling strategies, and benchmarking variables (e.g., pool size, initial seed, class balance ratios). It also includes information about the data sets used and the types of networks where the data sets were collected. Finally, we also inspected whether the studies

considered wrong labeling impact in their experiments.

Beaugnon et al. [18] focus on comparing their proposed sampling strategy, which is based on rare class detection, with other similar studies. This study lacks baseline comparison and elaboration on important benchmarking variables (e.g., pool size, initial seed). Its sampling strategy, balancing the training set with various malware families (i.e., noted as uncertainty with family annotation in Table 1), is compared to the uncertainty strategy as well as the proposals of other studies [22, 19]. Stokes et al. [19] apply an uncertainty sampling strategy supported by an anomaly detection model and compares the result of this approach with a method that only uses the uncertainty strategy. Görnitz et al. [22] utilize an active learning strategy to improve a semi-supervised anomaly detection method based on support vector data description. The sampling idea of this study benefits from the margin strategy (i.e., selecting the instances close to the decision hyperplane), which is supported by the identification of anomalous clusters. Despite demonstrating the effectiveness of an active learning-based approach, Li and Guo [10] do not elaborate on the optimization of this approach. Almgren and Jonsson [9] conduct experiments regarding the impact of pool size and variation of class balance in the data set.

All studies except ours address the usual IT networks without considering the impact of wrong labeling. Our study evaluates three active learning query strategies using different scoring measures and compares the performance with two baselines: a supervised model trained with a huge number of labeled data and a learning model induced by random query selection. Except for Almgren and Jonsson [9], the related research did not assess the impact of relevant variables on the implementation of active learning (e.g., seed size, pool size). Our study provides a detailed investigation of these variables.

4. Methodology

4.1. Data set

The data set used in this research is *MedBIoT* [24]. This data set is composed of benign and malicious traffic captured in a medium-sized IoT network architecture formed by real and virtual IoT devices. Malicious network data of three prominent IoT botnets are included in the data set (i.e., *Mirai* [25], *BashLite* [26], and *Torii* [27]). More specifically, the network data provided by *MedBIoT* are related to the initial steps of the botnet life cycle (i.e., spread and C&C [28]) for the three malware and do not include data related to attacking activities of the compromised IoT devices (bots). This fact could make the data set more complicated to categorize as the malicious traffic might not be as evident as usual attack traffic (i.e., characterized by a large volume of network packets and connections), so the malicious data might be more similar to benign IoT traffic (i.e., a lower volume of network data). In this study, all malicious labels (i.e., related to malware type) were abstracted to a single malware label (i.e., positive class) and the features used were the ones provided by *MedBIoT* in the structured data set format, as in [29]. In this regard, the feature set is composed of 100 data attributes based on statistical methods applied sequentially to the raw network packet data.In this study, it is assumed that the features are inherently interpretable by a human analyst (i.e., oracle). Given the large size of the original data set, a randomly selected sample of data points was used instead. More precisely, 150,000 data points were extracted from the data set files to compose a balanced data set. As a result, the data set was composed of 75,000 data points per class (i.e., malware and benign). This data set was split into two disjoint data sets for training and testing purposes, keeping the balanced class composition. The training data set was composed of 100,000 data samples, while the testing data set included 50,000 data points.

Even though hyper-parameter optimization of the learning models was not performed, the feature set was refined to remove redundant features (i.e., feature selection). Pearson's linear correlation coefficient was applied pair-wise to the feature set, and highly correlated features (i.e., Pearson's $r > 0.80$) were removed and not used as input data for the models.

The application of active learning requires the availability of a collection of unlabeled data (i.e., pool). For the purpose of this study, the pool of unlabeled data was always composed of data points extracted from the training data set (i.e., 100,000 samples). The testing data set was never used for training purposes and was always used to test the accuracy of the learning models at every iterative step in the active learning scenarios.

4.2. Baseline model (passive approach)

To proper evaluate the benefits of the active learning approach, a *passive* (i.e., *static*) baseline performance was required for comparison. In this study, *static* models were induced and taken as a baseline model. These models are trained using large labeled data sets and are rarely updated. They are representative of commonly used approaches to build intrusion detection models in production setups.

To establish a reference or baseline performance, the passive learning models induced were built using the same algorithm (i.e., a typical supervised model trained using a large data set) for proper comparison with the active learning results. In this regard, traditional ML algorithms (i.e., k-Nearest Neighbors, Decision Tree, Logistic Regression, Naive Bayes, Support Vector Machines, and Random Forest) were evaluated. The classification algorithm providing the best performance was selected for all the experimental scenarios.

The performance of the selected algorithm induced using different subsets of the training data and tested with the whole testing set was used as the baseline performance. These training data subsets of different sizes were selected randomly from the whole training data set.

Given the stochasticity of the data selection process and for the sake of the stability of the results, a total of 50 models were induced per training set size. The performance reported per model is the average of the 50 iterations. *Accuracy*, which reflects the ratio of correctly identified samples (i.e., malware or benign) on the testing set, was used as the models' performance metric.

The Study	Baselines	Sampling Strategies	Benchmarking Variables	Dataset(s)	Wrong Labeling	Target Network
[9]	Random query Greedy optimal	Uncertainty Balanced class selection	Pool size Class balance	Kdd Cup 1999 [23]	No	IT
[10]	Random query	Uncertainty	-	Kdd Cup 1999 [23]	No	IT
[18]	-	Uncertainty with family annotation Uncertainty Semi-supervised model [22] Rare category labeling [19]	-	NSL-KDD	No	IT
[19]	-	Uncertainty with anomaly detection Uncertainty	-	Kdd Cup 1999	No	IT
[22]	Random query	Margin strategy with anomaly detection Margin strategy	-	HTTP Traffic Data	No	IT
Our Study	Supervised model Random query	Uncertainty Query-by-Committee Ranked Batch-Mode	Initial seed Pool size Batch size	MedBIoT [24]	Yes	IoT

Table 1: Comparison of this study with similar studies

4.3. Active Learning Scenarios

As described in the previous section, the first step in this research was to find good baseline models that worked effectively on the classification task (i.e., early IoT botnet traffic detection). These *passive* models were used as comparative baselines for the performance of the active learning scenarios. The following scenarios were implemented to evaluate the active learning strategies explicated in Section 2:

- *Uncertainty sampling*: a single classifier (i.e., active learner) was used to generate an initial detection model that used the active learning cycle (see Figure 1) to update its knowledge based on different query instance selection criteria. The query instance selection criteria (i.e., scores) tested were *classification uncertainty*, *classification margin*, and *classification entropy*. The stopping criteria for the active learning cycle was 1,000 single queries, meaning that the process was repeated 1,000 times selecting a single sample per iteration. The accuracy provided by the models on the testing set at each training step was retrieved. In addition, the impact of two variables in the active learner performance was evaluated: the *size of the initial data set* (i.e., seed size), and the *size of the pool of unlabeled samples*.

- *Ranked batch-mode sampling*: a single classifier was used to generate an initial detection model that used the active learning cycle to update its knowledge using the *ranked-batch mode* query strategy, as defined in [15]. This enabled the learner to query for labeling more than a single instance per training step. The stopping criteria was the query of a total of 1000 instances. In this case, depending on the *batch* size, meaning how many instances were queried per training step, a variable number of training iterations were performed. For instance, 500 training steps were performed when a batch size of 2 query instances was used. The accuracy performance of the models on the testing set at each training step was retrieved. As in the previous scenario, the impact of the *pool size* and the *size of the initial seed* were evaluated. In addition, the impact of the *batch size* (i.e., the number of samples queried per training step) was assessed.

- *Query by committee*: a group of classifiers was generated, and the committee-based active learning strategy was used to update their knowledge based on different query instance selection scores. The scoring measures tested were *vote entropy*, *consensus entropy*, and *maximum disagreement*. Given the similarity between this active learning strategy and the single classifier strategy (i.e., *uncertainty sampling scenarios*), and to reduce the number of variables analyzed, only the *best* pool size of the uncertainty sampling scenarios was used for the committee-based scenarios. Therefore, for these scenarios, the impact of the *pool size* was not evaluated. However, the impact of the *initial seed size* and the *size of the committee* (i.e., number of members) were evaluated. The same stopping criterion as in the uncertainty sampling scenarios was applied (i.e., 1000 single instance training steps). The accuracy performance of the models on the testing set at each training step was retrieved.

In the previous scenarios, it was assumed that the *oracle* labeling the queries provided a 100% accuracy on class identification, that is, a 0% error in data class imputation. Although desirable, this scenario is not realistic. For instance, in SOC environments, where analysts with different degrees of experience and expertise coexist, even though a high degree of accuracy can be expected, a certain *margin* of error cannot be ruled out. For this reason, the last kind of scenarios evaluated considered the possibility of wrong labeling in the active learning implementation. The *possibility* of wrong labeling was materialized assigning a wrong labeling probability to the scenarios. More specifically, for each queried sample (i.e., the most informative sample according to the used strategy), a wrong labeling probability (e.g., 5%) was implemented using a random number generator function. If the number was below an established threshold related to a probability, the label was flipped.

To implement the wrong labeling scenarios, the best models from the previous scenarios were selected (i.e., one per strategy), and different wrong labeling probabilities were evaluated. Furthermore, the baseline models, built using the *passive learning* approach, were also induced implementing the wrong labeling scenario. In this case, the wrong labeling probability function was applied to each data sample in the training set before training the model.

All scenarios, including active and passive learning approaches, were tested on the same testing data set. Besides, as all scenarios involved some degree of randomness, 50 iterations were performed per scenario. Therefore, the reported performance of the models is the average accuracy score of the 50 iterations. As both the training and testing data sets are balanced, the accuracy score provided a reliable and comprehensive score of the models' performance. The implementation of the active learning strategies was performed using the *modAL* library for Python [30].

5. Results

This section provides the main results of the experimental setup where the active learning scenarios were tested.

5.1. Data preprocessing

The data preprocessing stage aimed to remove redundant and irrelevant data features. For this purpose, Pearson's linear correlation coefficient (r) was applied. Features highly correlated with any other feature were removed ($|r| > 0.80$), keeping just one of them. As a result, from the initial set of 100 features used to describe every sample of the data set, only 20 features remained in the feature set.

For the purpose of data visualization, t-Distributed Stochastic Neighbor Embedding (t-SNE) [31] technique was used. t-SNE is a dimensionality reduction technique that enables the visualization of high dimensional data sets in low dimensional spaces. It captures the structure of the data in a way that neighboring points in the high dimensional space are likely to be neighboring points in the low dimensional space representation. Therefore, it enabled us to get a sense of the data structure as depicted in Figure 2.

As can be observed in the two-dimensional representation of the data set in Figure 2, the classes seem to be spread and clustered in different regions of the feature space (i.e., red refers to malware data and green to benign data). This may indicate relatively good separability of the data. However, some areas show completely mixed data, which can hinder class discrimination in those regions and, consequently, the application of the active learning approach.

5.2. Machine learning algorithm selection

All the scenarios tested in this study used the same base algorithm and hyper-parameters, which ensured *equal* conditions for the active and passive learning scenarios. To assess the goodness of different machine learning algorithms for

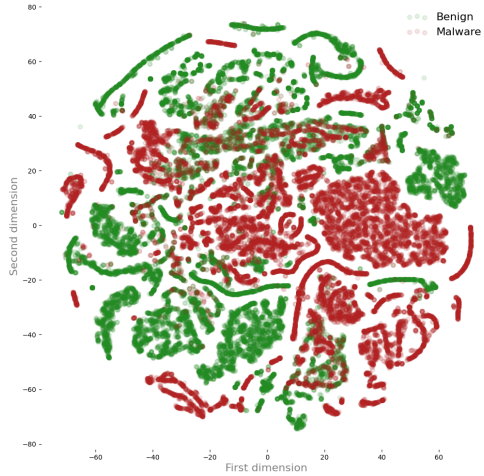


Figure 2: Comparison of algorithms performance

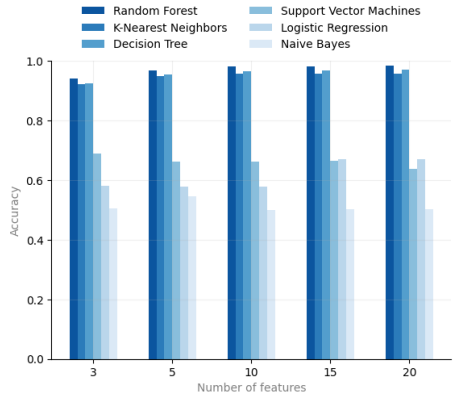


Figure 3: Comparison of algorithms performance

the task and select the best one for the scenarios, classification models were induced using different subsets of features and traditional machine learning classification algorithms (i.e., *k-Nearest Neighbors*, *Decision Tree*, *Random Forest*, *Support Vector Machines*, *Logistic Regression* and *Naive Bayes*). The features were ranked according to discriminatory power based on Fisher's score [12]. The feature subsets included the top n features based on the ranked list of features, where n was an integer (i.e., 3, 5, 10, 15, 20). The performance results for all the induced models are provided in Figure 3. The whole training set was used to generate the classification model, and the whole testing set was used to evaluate its predictive performance.

As can be observed in Figure 3, the Random Forest (RF) algorithm outperformed all the other algorithms in all cases. Therefore, the RF algorithm was selected for all the benchmarking scenarios using the default hyper-parameters of the *scikit-learn* library for Python [32].

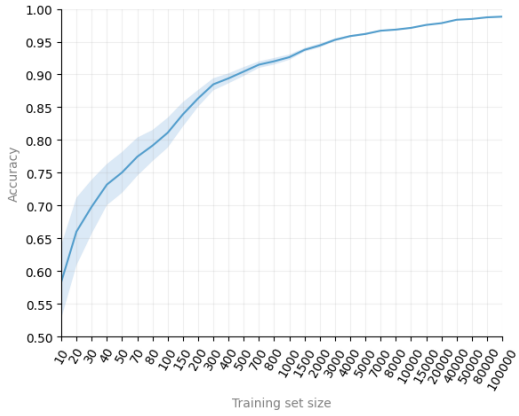


Figure 4: Baseline model on distinct training subsets

5.3. Baseline model performance

The baseline classification algorithm, Random Forest, was used to build classification models with training data subsets of different sizes. For this purpose, training subsets of different sizes were randomly sampled from the whole training set without replacement. For all training set sizes, 50 models were induced. The whole testing set was used to test the predictive performance of the models. The results reported are the average and standard deviation of the 50 models per training subset. The results are provided in Figure 4. In this line graph, the horizontal axis provides the training data set size, and the vertical axis the average accuracy performance obtained on the testing set. The blue line provides the average value, while the standard deviation is reported with the surrounding colored area. For the sake of interpretation, the vertical axis range was limited to the observed values, thus restricted to the [0.5, 1] interval.

As shown in Figure 4, the larger the training data size, the greater the accuracy retrieved and the lower the variability observed among iterations. Furthermore, the performance line is steeper for the smaller data sets, suggesting that the model could learn effectively from a small training data set and increase its knowledge with a small number of added samples. For instance, from 10 to 40 data samples, the performance increased from 0.58 to 0.73, whereas from 10,000 to 40,000 samples, the increase was barely 0.01, from 0.97 to 0.98. The highest performance obtained was of 0.988 accuracy (i.e., 98.8%) with 100,000 samples, that is, using the whole training set. The 0.90 accuracy performance was achieved with 500 samples, whereas 0.95 and 0.97 accuracy values were achieved using data sets containing 3,000 and 10,000 samples, respectively.

5.4. Active learning scenarios

The following paragraphs report the performance results for the active learning scenarios described in Section 4.

5.4.1. Uncertainty sampling

The three query instance selection scoring measures, namely *classification uncertainty*, *classification margin*, and *classification entropy*, were evaluated under the same conditions, and their performance. The impact of using a different number of samples in the unlabeled pool of data (i.e., *pool size*) and the number of samples used for the initial model (i.e., *seed size*) was assessed. The initial seed size tested were 2, 10, 40, and 200 samples. In all cases, the training set was balanced. This implied that if the initial seed was 2 instances, one sample per class was selected randomly from the whole training set. Random selection of the initial seed data ensured the generation of a different model per iteration. Similarly, for the pool size, each specific active learning model scenario was implemented using different unlabeled pool sizes. The instances that composed each pool for each scenario were randomly selected. Given the randomness of the scenarios and for the sake of stability of the results, 50 models were induced per combinatorial scenario. The reported result per combination is the average performance of all the models induced per scenario. The results for the scenarios using *classification uncertainty* as query selection metric are provided in Figure 5.

Initially, it seems logical to assume that, for both variables, the larger the number of instances, the better the performance. Therefore, the classifier should be trained initially with the maximum labeled data available as the initial seed, and the unlabeled pool should contain the maximum number of unlabeled instances available. In this regard, in our testbed, we assumed that the maximum number of labeled instances available was 200, and the maximum number of unlabeled samples in the pool was 100,000.

However, as can be observed in Figure 5, in all graphs (i.e., using any seed size), the maximum accuracy performance is reached with an unlabeled pool of about 7,000 instances (i.e., red line). The graphs evidence that the learning is hindered when the unlabeled pool is too small (i.e., lightest green line) or too big (i.e., darkest blue line). In the former case, the pool might not be representative of the underlying data distribution in most iterations, and, consequently, even though the learning process works well in the initial steps, it is halted when the representative data is exhausted, as evidenced by the plateau in performance. In the latter case, the existence of many similar instances for which the uncertainty scores might be similar or identical may hinder the selection of the optimal learning sample, leading to slower learning. This phenomenon is evidenced by the slow pace of learning observed in this scenario, which reaches the lowest performance of all pool sizes , and shows the smallest slope and no signs of flattening (i.e., the model was still learning).

Regarding the seed size, it is worth noticing that all models' performance is over 0.97 regardless of the initial training data set used. The only substantial difference observed among the models is the shape of the curve leading from the initial model to the last query. As can be observed in Figure 5, except for the model with an initial seed of 2 instances (i.e., top-left graph), the models suffer from an initial decrease in performance, which is corrected later with an increase in performance. This correction takes more time (i.e., queries) for the models induced with a larger initial seed size. This initial performance decrease might have been caused by the initial bias

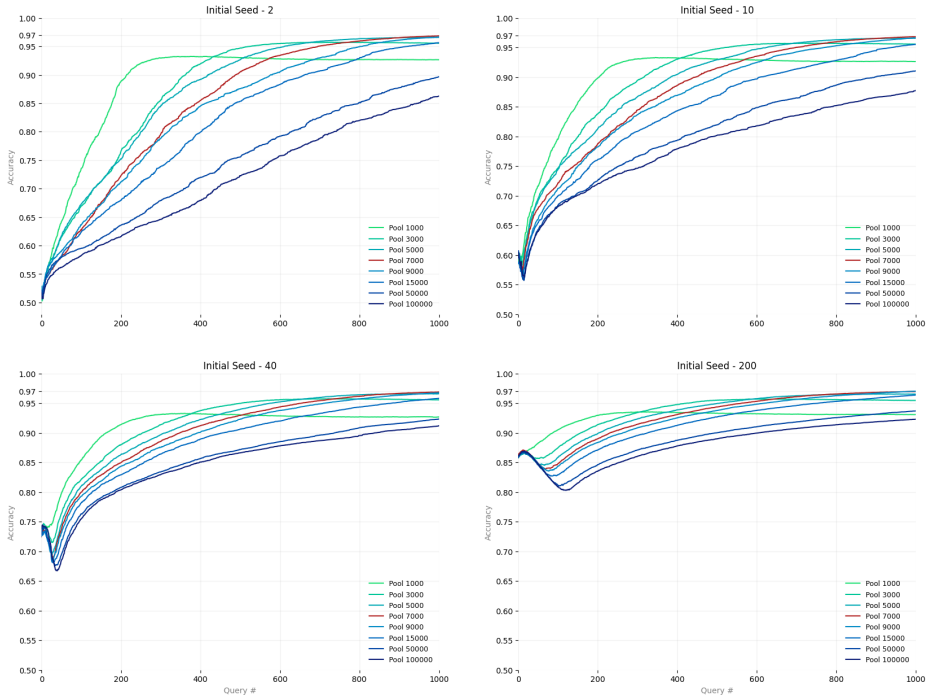


Figure 5: Uncertainty sampling: classification uncertainty results

of the classifiers induced, thus selecting sub-optimal instances in the first iterations that may respond to the model bias but not provide better generalization capabilities. This fact is emphasized for larger pools. In any case, all the active learning models recover from the initial draw-down, surpassing the 0.95 performance in about 600 queries and achieving 0.97 in around 800 queries.

Comparatively, the baseline models (i.e., the passive learning models depicted in Figure 4) reached 0.95 and 0.97 accuracy scores with 3,000 and 10,000 samples, respectively. This shows that the active learner can achieve similar performance to the passive model using 10 to 12 times fewer labeled samples. Besides, the active learning approach seems to provide additional benefits when the initial training set (i.e., initial seed) is small, as no initial draw-down in performance is observed in that case. However, as a trade-off, the initial accuracy is lower when the initial seed is smaller.

In the uncertainty sampling scenarios, the three instance scoring measures evaluated produced similar results. A comparison of the performance of the three metrics (i.e., classification uncertainty, margin, and entropy) using a pool size of 7000 and the four initial seed sizes are provided in Figure 6. More interestingly, this figure includes the performance of the random query selection approach, where the query sample is selected randomly from the unlabeled pool. This approach is provided as a comparative baseline that can be used to evaluate the effectiveness of the query selection metrics. In this

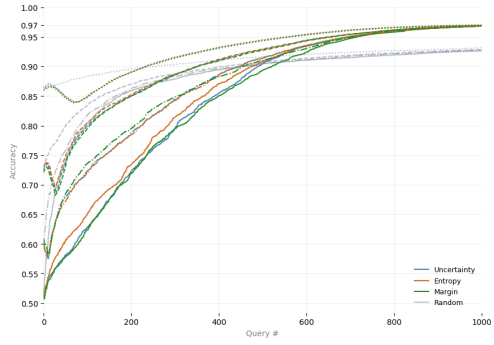


Figure 6: Comparison of uncertainty sampling strategies and random query selection

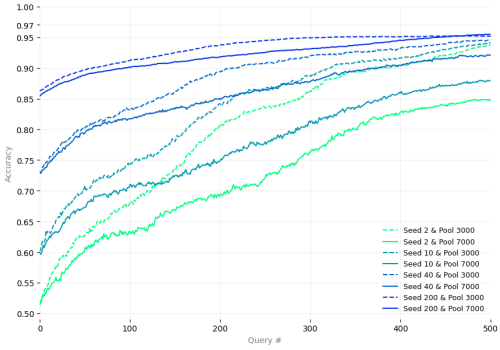


Figure 7: Ranked batch-mode sampling performance for batches of 2 samples per query

regard, the random approach seems to provide better performance than the active learning approaches in the initial stages. However, as the number of queries increases, the active learner approach learns better and faster than the random query selection, achieving higher accuracy, for any initial seed size, after 600 queries. Besides, the maximum performance of the random instance selection strategy is lower and plateaus faster than any of the active learning query strategies. More precisely, the random approach achieved a maximum accuracy of 0.93 after 1000 queries in the best model, whereas this performance was reached by most active learners in 500 queries, achieving over 0.97 accuracy score after 1000 queries to the oracle.

5.4.2. Ranked batch-mode sampling

The classic pool-based active learning strategy, uncertainty sampling, limits the query size to a single instance. Ranked batch-mode sampling was designed to overcome this limitation enabling the learner to query multiple instances per query and enhancing the informativeness score using an additional scoring metric, as detailed in Section 2. The ranked batch-mode sampling strategy was implemented and tested in our experimental setup. For a proper evaluation of this strategy, different batch sizes (i.e., 2, 3, 4, 5, 7, 10, 20) were used in addition to different pool sizes and initial seeds.

Based on the results, the scoring proposed by this query strategy did not provide any additional improvement. The ranked batch-mode strategy did not provide better performance than the uncertainty sampling strategy. As a reference, Figure 7 provides the performance results for the batch size of 2 instances which implied 500 training steps. The graph also provides the different seed sizes (i.e., SX, where X specifies the size) and pool sizes evaluated (i.e., specified in the legend as 3,000 or 7,000 next to the seed size, separated by a hyphen).

Despite the low performance, an interesting result from the ranked batch-mode scenarios is that the best performance was achieved using a smaller pool size than for uncertainty sampling, achieving the maximum accuracy performance of 0.95 with an unlabeled pool of 3,000 instances. As can be seen in Figure 7, the pool of 7,000 unlabeled instances provided significantly lower results. Besides, even though the ranked batch-mode models did not provide improved performance regarding the single query mode, none of the models in Figure 7 show the performance decrease in the initial stages observed for uncertainty sampling, as shown in Figure 5 and Figure 6.

These results suggest that a hybrid approach, combining uncertainty-based active learners with random selection or ranked batch-mode only for the initial queries, may help to overcome the initial reductions in performance and bias evidenced by the uncertainty sampling models. This could avoid the draw-downs and provide sustained learning leading to high-performance results.

5.4.3. Query by committee (QBC)

The previous scenarios used a single learner to select the most informative instances and pose queries to the oracle. In the query by committee approach, a group of learners trained on the same data but representing competing hypotheses decide about the most informative instances based on their disagreement. The labeled instance is incorporated into each committee member's training set. The query strategies tested in our testbed were *vote entropy*, *consensus entropy* and *max disagreement*, as explicated in Section 2. As this approach is very similar to the uncertainty sampling strategy (i.e., the decision is based on committee disagreement but the individual models behave similar to the uncertainty sampling classifiers), the best pool size for the uncertainty sampling strategy was used for the committee-based scenarios (i.e., 7,000 samples). The impact of the size of the committee and the initial seed size on the performance of the QBC models was explored.

The results for the *consensus entropy* query instance selection metric are provided in Figure 8. Similar to Figure 5, each graph in Figure 8 shows the performance of the models built with different initial seeds (i.e., 2, 10, 40, and 200). For each model, different committee sizes were tested (i.e., 2, 3, 5, 7, and 10). They are reported using different colors, and designated as *CE* in the legend. For the sake of comparison, the best model for the *maximum disagreement* (i.e., MD) and *vote entropy* (i.e., VE) metrics are provided using red lines and dashed and solid line styles, respectively. The *MD* and *VE* scoring metrics provided significantly lower performance than the CE strategy. Therefore, their complete results are not reported.

As can be observed in Figure 8, the larger the committee size, the better the results. The largest committee shows the steepest learning curve in all cases (i.e., the fastest learning). Besides, a large committee size tends to avoid the decrease in performance on the initial queries and provides improved learning in the early stages of the active learning process. However, a larger committee implies retraining more models after the labeling process, which might be more time-consuming, and demand more resources. In any case, using the *CE* scoring metric, a 0.95 accuracy is achieved before 600 queries and 0.97 before 800 queries, regardless of the initial seed size.All models converge beyond 900 queries, providing the same final performance.

The *MD* and *VE* scoring metrics showed sub-optimal performance, especially MD, where the best model, using a commit-

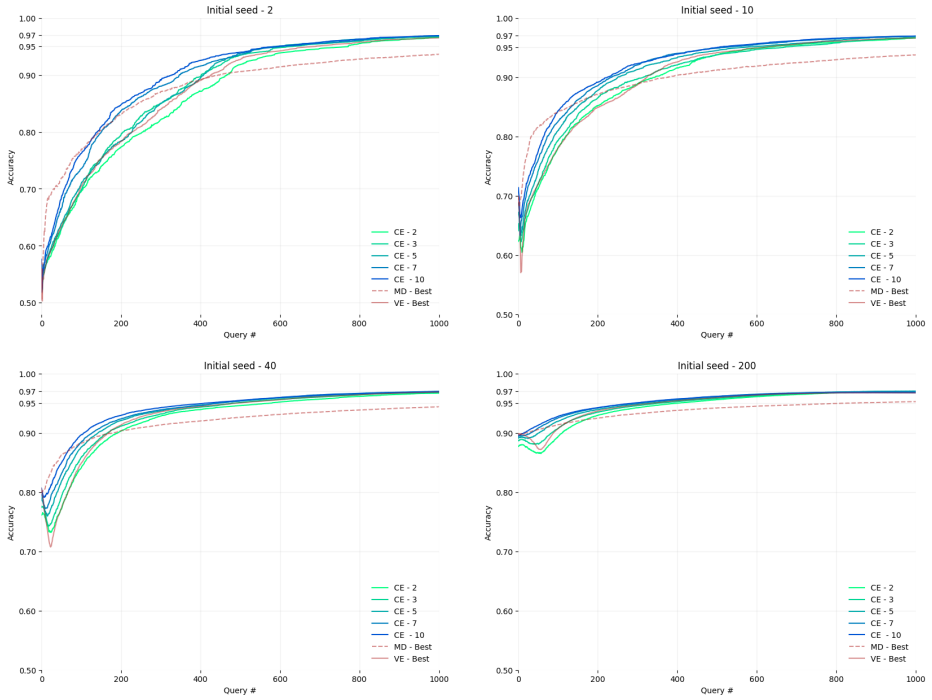


Figure 8: Query by committee performance results

tee of 10 members, did not even achieve 0.95. The best VE model, built with 10 members, reached 0.97 performance, but its learning curve is worse than CE for 5 members. Besides, the VE strategy suffered from a pronounced decrease in performance in the initial queries for all models.

Given the similarities between uncertainty sampling and query by committee (QBC), a direct comparison between both query strategies is well-motivated. In this regard, Figure 9 provides the comparison between both active learning strategies for different initial seeds. The best uncertainty model and two QBC models (i.e., 3 and 10 members) are provided.

As can be seen in Figure 9, both active learning strategies converge to 0.97, which shows the goodness of either of the approaches to achieve high performance with a small fraction of the data needed by the passive learning approach (see Figure 4). However, the learning curves of both strategies are notably different, as depicted in Figure 9, especially for a small number of initial seed (i.e., larger separation between the curves). In any case, the QBC strategy with a committee of 10 classifiers provides the steepest learning curve , achieving high-performance faster than the other approaches. Besides, the initial models for the QBC strategy start at a higher accuracy score than the uncertainty sampling models. Therefore, the *ensemble* of classifiers that form the committee provides improved performance from the initial step, emphasizing the goodness of combining several classifiers for enhanced learning. However, as mentioned before, implementing committee-based strategies require more

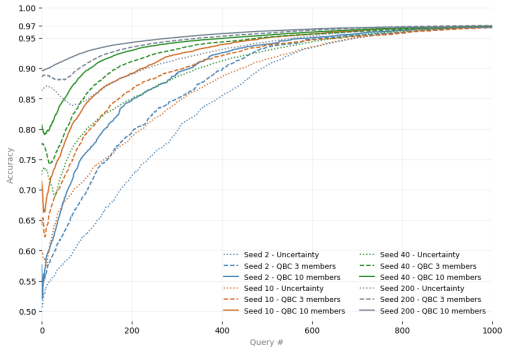


Figure 9: Committee vs uncertainty sampling

resources and may imply longer retraining schedules, especially when the committee size is large. For this reason, the QBC with a committee size of 3 may be a good trade-off between the best QBC model, composed of 10 classifiers, and the best uncertainty model (i.e., using a single classifier).

5.4.4. Wrong labeling impact

The previous scenarios assumed a perfect labeling approach, that is, a 0% error in the label provided by the oracle. Even though this is highly desirable, it is unrealistic due to the possible limitations of the oracles (e.g., inexperienced analysts) and the inherent difficulties of the data labeling process (e.g., anomalous instances or confusing data attributes). For these reasons, it is worth considering the case of labeling mistakes.

The best models from the previous scenarios were used to implementing a probabilistic wrong labeling approach. In this regard, different wrong labeling probabilities (i.e., 5%, 15%, and 25%) were applied to every query instance. Figure 10 shows the results for the best uncertainty sampling model (i.e., using classification uncertainty), the best QBC model (i.e., CE with a committee of 10 members), and the best ranked batch-mode model (i.e., batch of 2 instances) for different initial seed sizes (i.e., 2, 10, 40, and 200) and different wrong labeling probabilities (i.e., 0%, 5%, 15%, and 25%). For the sake of comparison, the wrong labeling probability was also applied to the training set of the passive models. As in the previous scenarios, 50 iterations were performed per model for the sake of stability and representativeness of the results.

As can be observed in Figure 10, the active learners with the smaller initial seeds are the ones affected the most by the wrong labels.

This is expected as for the active learners, the wrong labeling is applied to the instance selected as optimal for learning purposes, the most informative among the instances in the unlabeled pool. Therefore, this fact may produce a notable bias, affecting the results significantly. However, despite that, the active learners still show significant improvement and good learning output, even in the extreme case of the 25% wrong labeled samples. Besides, the graphs show that when the seed size is relatively large, the models are remarkably resilient to the 5% wrong labeling probability, providing similar final performance to the perfect labeling approach (i.e., 0% error). Based on these graphs, wrong labeling below 15% has a minimal impact on the models, yielding good performance over time. Larger proportions may significantly affect the performance of the models. This fact is also applicable to the passive models.

6. Discussion

The application of active learning is especially suitable for those domains where large amounts of unlabeled data are available, and the labeling cost might be a limiting issue. In this research, we performed a thorough benchmarking of the active learning approach for the IoT botnet detection problem domain, simulating a SOC environment where human experts are available.

The results show that active learners can provide the same performance as passive learning approaches using significantly fewer data (i.e., at least 10 times fewer labeled samples) and that the *smart* query implementation performed by the active learning approaches outperforms the random selection of instances in the long term. Therefore, in those scenarios where a wealth of unlabeled data is available but the labeling cost is expensive, the active learning approach can provide great benefits, optimizing resources and minimizing the labeling cost as less labeled data is required by the active learners to yield high performance. Furthermore, the active learners with a small seed size (i.e., initial training set) may provide better learning capabilities and minimize the labeled data needed to achieve performance similar to models trained initially with a larger quantity of labeled data. However, models using small seed sizes are more affected by mislabeled instances. Thus a large initial seed is recommended to increase the initial robustness of the model if wrong labeling is expected to be a significant issue.

In any case, the active learning approach is further enhanced when the predictive power of an *ensemble* of classifiers is leveraged. However, this implementation requires more resources which might be a restricting issue in resource-limited environments.

On the other side, as shown in Figure 6, random sampling provided better results than uncertainty sampling for the first few hundred queries but did not achieve higher detection performance in the later stages. This outcome may indicate *sampling bias* which causes a decrease in the effectiveness of the uncertainty sampling strategies. Although these strategies do not get stuck at low-performance levels, it can be deduced that, due to the initial model bias, the capability of the initial queries to explore the feature space is relatively limited. In some applications, the proper exploration of the feature space might be essential for generating an accurate decision boundary. The difference between random sampling and the active learning strategies decreases when a larger initial seed is selected, meaning that more instances in the initial training set may provide better space exploration. Nevertheless, an active learning-based solution that utilizes random sampling for the first set of queries or mixes random sampling with other sampling strategies may yield enhanced performance for the active learning approach.

The ranked batch-mode sampling strategy resorts to uncertainty-based methods, which are prone to sampling bias. However, this approach tries to avoid getting stuck in previously-explored regions of the feature space by calculating the similarity of each unlabeled instance with the samples already incorporated into the model [15]. However, in our study, the additional scoring proposed by this method to select the optimal learning query did not improve the results yielded by the uncertainty sampling strategies.

Most research focuses on perfect labeling , thus assuming no error in the labeling process. Even though this would be ideal, this assumption is highly unrealistic. For the sake of completeness, our benchmarking explored the impact of wrong labeling on the active learning implementation and the generation of the passive models. The results show that active learning models are resilient to low error rates, which are the expected

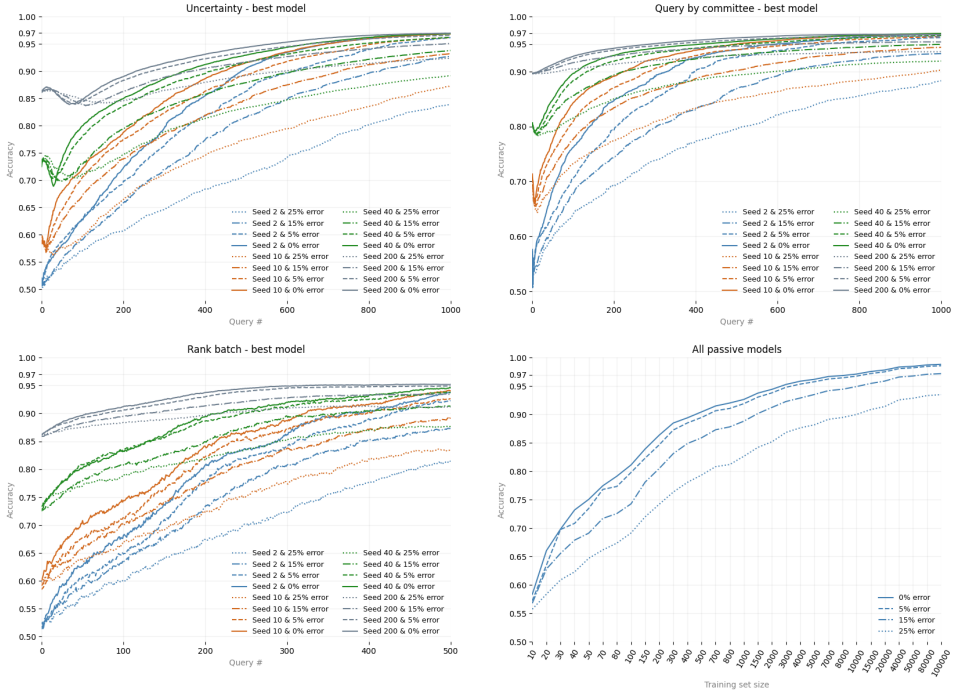


Figure 10: Wrong labeling impact

rates when domain experts act as oracles. For high error rates (i.e., over 15%), the active learning approach may lead to sub-optimal performance.

The noisy or wrong labeling problem has been studied in classification problems in which the labeling cost is relatively cheap (i.e., non-expert labeling cases), and a repetitive labeling strategy works as a solution [33]. Estimating the accuracy of noisy labelers and incorporating this knowledge into the multiple labeler context was studied in [34]. Intrusion detection requires experts in their fields, meaning that the expected wrong labeling ratio should be considerably low. According to our results, the error ratios of around 15% would induce models that are reasonably resistant to the labeling problems that can be experienced in the field. It is important to note that we assume that the probability of wrong labeling is equal for each query. However, this might not be completely accurate as experts with varying knowledge and experience are involved in the tier structures of SOCs. Therefore, the probability of mislabeling may differ depending on the expert who receives the query. Another variable that deserves attention in this context is the variation of the labeling cost, as a less fallible expert may have a higher associated cost than an inexperienced analyst. In this regard, it would be an interesting research direction to consider the cost variations with wrong labeling.

In summary, our benchmarking study demonstrates the goodness of the active learning approach in the analyzed scenarios and that the human-machine interaction may yield high-performance results with significantly fewer data.

7. Limitations and Threats to Validity

Our work aims to provide an initial benchmarking of the active learning approach to generate more efficient and effective intrusion detection systems (IDS) for IoT networks. As an initial approach and novel application to the field, it does not aim to convey a complete solution for the phenomenon but to assess the feasibility of the active learning framework in generating better IDS in the IoT context. We also evaluated the impact of different variables (e.g., pool and seed size) in the implementation, performing a comprehensive benchmarking. Based on that, our work is not free from threats to the validity of our results and limitations, which are described as follows.

- *Data set*: The IoT environment is a complex network infrastructure characterized by constant dynamism, resource constraints, and large data volumes, among other features. The data set used in this study, *MedBIoT*, is one of the larger and more realistic IoT botnet data sets, which captures part of this challenging environment and enables the simulation of benign and malicious network behavior of a medium-sized IoT architecture. In this regard, we performed an empirical study using a specific data set. However, the learning models induced in this study might not be directly transferable to some IoT network architectures

(i.e., larger IoT networks, variable types of benign traffic). Our results might be comprehended as an initial comprehensive evaluation of the active learning approach to induce better IDSs. More empirical studies are needed for replicating and exploring the results on other data sets that capture the characteristics of other IoT architectures, which is in line with our future work.

- *Data features*: In our benchmarking, we assumed that the data features were interpretable for experts. For this reason, we did not explore the implications and relation of specific features in the discriminatory process. As the main objective of this work was the exploration of active learning, we did not cover other relevant issues related to the interaction of SOC experts with the model and the nature of incident handling operations at SOCs (e.g., teams are usually composed of members with different levels of expertise, so labeling costs and wrong labeling ratios may change among members), which were out of the scope of this paper. However, they constitute part of our future work in the problem domain.

- *Balancing data*: In our benchmarking, we assumed that the data set (i.e., training and testing sets) were balanced. The training set was balanced to avoid biased classifiers, a critical aspect to consider for high-performance models. The testing set was balanced to make the accuracy metric meaningful as a performance metric. The imbalanced data issue might be a significant variable to address in real IoT networks as in normal operation, most of the traffic should be benign. This aspect was not included in our paper for the sake of comprehensibility of this exploratory paper. Therefore, it was out of the scope of this paper but remains as future work on the topic.

8. Conclusions

The application of active learning for IoT detection in the SOC context is well-motivated due to the large amount of unlabeled data generated in those environments and the availability of human experts for labeling purposes. The *smart* instance query selection strategy performed by the active learning strategies can help to build and enhance the detection models using significantly less data than the traditional models used to build machine learning-based detection systems (i.e., *passive* approaches). Of the general active learning approaches tested in our benchmarking, the QBC-based strategies yield highly accurate detection models with at least 10 times less data than passive approaches for model generation. If resources are limited, uncertainty sampling strategies can generate similar models, but they may require a larger quantity of labeled data. Besides, our results show that active learning models are resilient to wrong-labeled data when the proportion of mislabeled instances is relatively low (i.e., below 15%), expected in SOC environments where domain experts act as oracles.

References

[1] S. S. Silva, R. M. Silva, R. C. Pinto, R. M. Salles, Botnets: A survey, Computer Networks 57 (2013) 378–403.

[2] Cisco, Cisco annual internet report (2018-2023) white paper, 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[3] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, Y. Zhou, Understanding the mirai botnet, in: Proceedings of the 26th USENIX Security Symposium, 2017, pp. 1093–1110.

[4] Ponemon Institute LLC, The cost of malware containment, 2015.

[5] A. L. Buczak, E. Guven, A survey of data mining and machine learning methods for cyber security intrusion detection, IEEE Communications surveys & tutorials 18 (2015) 1153–1176.

[6] H. Bahşi, S. Nõmm, F. B. La Torre, Dimensionality reduction for machine learning based iot botnet detection, in: 2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV), IEEE, 2018, pp. 1857–1862.

[7] S. Nõmm, H. Bahşi, Unsupervised anomaly based botnet detection in iot networks, in: 2018 17th IEEE international conference on machine learning and applications (ICMLA), IEEE, 2018, pp. 1048–1053.

[8] H. Schütze, E. Velipasaoglu, J. O. Pedersen, Performance thresholding in practical text classification, in: Proceedings of the 15th ACM international conference on Information and knowledge management, 2006, pp. 662–671.

[9] M. Almgren, E. Jonsson, Using active learning in intrusion detection, in: Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004., IEEE, 2004, pp. 88–98.

[10] Y. Li, L. Guo, An active learning based tcm-knn algorithm for supervised network intrusion detection, Computers & security 26 (2007) 459–467.

[11] T. Riebe, T. Wirth, M. Bayer, P. Kühn, M.-A. Kaufhold, V. Knauthe, S. Guthe, C. Reuter, Cysecalert: An alert generation system for cyber security events using open source intelligence data, in: International Conference on Information and Communications Security, Springer, 2021, pp. 429–446.

[12] C. C. Aggarwal, Data mining: the textbook, Springer, 2015.

[13] B. Settles, Active learning literature survey (2009).

[14] B. Settles, M. Craven, An analysis of active learning strategies for sequence labeling tasks, in: proceedings of the 2008 conference on empirical methods in natural language processing, 2008, pp. 1070–1079.

[15] T. N. Cardoso, R. M. Silva, S. Canuto, M. M. Moro, M. A. Gonçalves, Ranked batch-mode active learning, Information Sciences 379 (2017) 313–337. URL: https://www.sciencedirect.com/science/article/pii/S0020025516313949. doi:https://doi.org/10.1016/j.ins.2016.10.037.

[16] B. Settles, From theories to queries: Active learning in practice, in: Active learning and experimental design workshop in conjunction with AISTATS 2010, JMLR Workshop and Conference Proceedings, 2011, pp. 1–18.

[17] J. Long, J.-P. Yin, E. Zhu, W.-T. Zhao, A novel active cost-sensitive learning method for intrusion detection, in: 2008 International Conference on Machine Learning and Cybernetics, volume 2, IEEE, 2008, pp. 1099–1104.

[18] A. Beaugnon, P. Chifflier, F. Bach, Ilab: An interactive labelling strategy for intrusion detection, in: International Symposium on Research in Attacks, Intrusions, and Defenses, Springer, 2017, pp. 120–140.

[19] J. W. Stokes, J. Platt, J. Kravis, M. Shilman, Aladin: Active learning of anomalies to detect intrusions (2008).

[20] N. Nissim, A. Cohen, Y. Elovici, Aldocx: detection of unknown malicious microsoft office documents using designated active learning methods based on new structural feature extraction methodology, IEEE Transactions on Information Forensics and Security 12 (2016) 631–646.

[21] S. D. Bhattacharjee, A. Talukder, E. Al-Shaer, P. Doshi, Prioritized active learning for malicious url detection using weighted text-based features, in: 2017 IEEE International Conference on Intelligence and Security Informatics (ISI), IEEE, 2017, pp. 107–112.

[22] N. Görnitz, M. Kloft, K. Rieck, U. Brefeld, Toward supervised anomaly detection, Journal of Artificial Intelligence Research 46 (2013) 235–262.

[23] U. K. Archive, Kdd cup 1999 data, http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, 1999.

[24] A. Guerra-Manzanares, J. Medina-Galindo, H. Bahsi, S. Nõmm, Medbiot: Generation of an iot botnet dataset in a medium-sized iot network., in: ICISSP, 2020, pp. 207–218.

[25] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al., Understanding the mirai botnet, in: 26th {USENIX} security symposium ({USENIX} Security 17), 2017, pp. 1093–1110.

[26] A. Marzano, D. Alexander, O. Fonseca, E. Fazzion, C. Hoepers, K. Steding-Jessen, M. H. Chaves, Í. Cunha, D. Guedes, W. Meira, The evolution of bashlite and mirai iot botnets, in: 2018 IEEE Symposium on Computers and Communications (ISCC), IEEE, 2018, pp. 00813–00818.

[27] Avast, Torii botnet - not another mirai variant, https://blog.avast.com/new-torii-botnet-threat-research, 2018.

[28] N. Hachem, Y. Ben Mustapha, G. G. Granadillo, H. Debar, Botnets: Lifecycle and taxonomy, in: 2011 Conference on Network and Information Systems Security, 2011, pp. 1–8. doi:10.1109/SAR-SSI.2011.5931395.

[29] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, A. Shabtai, D. Breitenbacher, Y. Elovici, N-baiot—network-based detection of iot botnet attacks using deep autoencoders, IEEE Pervasive Computing 17 (2018) 12–22.

[30] T. Danka, modal: A modular active learning framework for python3, https://modal-python.readthedocs.io/en/latest/, 2021.

[31] L. van der Maaten, G. Hinton, Visualizing data using t-sne, Journal of Machine Learning Research 9 (2008) 2579–2605. URL: http://jmlr.org/papers/v9/vandermaaten08a.html.

[32] scikit learn, sklearn.ensemble.randomforestclassifier, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html, 2022.

[33] V. S. Sheng, F. Provost, P. G. Ipeirotis, Get another label? improving data quality and data mining using multiple, noisy labelers, in: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, 2008, pp. 614–622.

[34] P. Donmez, J. G. Carbonell, J. Schneider, Efficiently learning the accuracy of labeling sources for selective sampling, in: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, 2009, pp. 259–268.

Curriculum Vitae

1. Personal data

Name	Alejandro Guerra Manzanares
Date and place of birth	19 April 1989 Suria, Barcelona, Spain
Nationality	Spanish

2. Contact information

Address	Tallinn University of Technology, School of Information Technologies, Department of Software Science, Centre for Digital Forensics and Cyber Security, Ehitajate tee 5, 19086 Tallinn, Estonia
Phone	+34 669 12 79 63
E-mail	alejandro.guerra@taltech.ee

3. Education

2018–…	Tallinn University of Technology, School of Information Technologies, Information and Communication Technology, PhD studies
2017–2018	Tallinn University of Technology, Faculty of Information Technologies, Cyber Security, specialty in Digital Forensics, MSc *cum laude*
2013–2017	Polytechnic University of Catalonia, EPSEM, ICT Engineering, BSc *cum laude*
2009–2013	Autonomous University of Barcelona, Faculty of Law, Criminology, BSc *cum laude*

4. Language competence

Spanish	native
Catalan	native
English	fluent
Estonian	basic

5. Honours and awards

- 2018, Estonian Research Council Master's Thesis Contest. 3rd Prize award winner. Category of Natural Sciences and Engineering.

6. Defended theses

- 2018, *Application of full machine learning workflow for malware detection in Android on the basis of system calls and permissions*, MSc, supervisors Prof. Hayretdin Bahsi and Dr. Sven Nõmm, Tallinn University of Technology
- 2017, *HoneyIo4 - The construction of a virtual, low-interaction IoT Honeypot*, BSc, supervisors Dr. Francisco del Aguila Lopez and Prof. Hayretdin Bahsi, Polytechnic University of Catalonia
- 2013, *Análisis documental de la interceptación de las telecomunicaciones en España*, BSc, supervisor Dr. Francesc Xavier Moreno Oliver, Autonomous University of Barcelona

7. Scientific work
Papers

1. A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. In-depth feature selection and ranking for automated detection of mobile malware. In *Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*, pages 274–283. INSTICC, SciTePress, 2019

2. A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Differences in android behavior between real device and emulator: A malware detection perspective. In *Proceedings of the 6th International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 399–404, 2019

3. A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. Time-frame analysis of system calls behavior in machine learning-based mobile malware detection. In *International Conference on Cyber Security for Emerging Technologies (CSET)*, pages 1–8, 2019

4. A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Kronodroid: Time-based hybrid-featured dataset for effective android malware detection and characterization. *Computers & Security*, 110:102399, 2021

5. A. Guerra-Manzanares, M. Luckner, and H. Bahsi. Android malware concept drift using system calls: Detection, characterization and challenges. *Expert Systems with Applications*, 206:117200, 2022

6. A. Guerra-Manzanares, M. Luckner, and H. Bahsi. Concept drift and cross-device behavior: Challenges and implications for effective android malware detection. *Computers & Security*, 120:102757, 2022

7. A. Guerra-Manzanares and M. Välbe. Cross-device behavioral consistency: Benchmarking and implications for effective android malware detection. *Machine Learning with Applications*, 9:100357, 2022

8. A. Guerra-Manzanares, H. Bahsi, and M. Luckner. Leveraging the first line of defense: A study on the evolution and usage of android security permissions for enhanced android malware detection. *Journal of Computer Virology and Hacking Techniques, in press*, 2022

9. A. Guerra-Manzanares and H. Bahsi. On the relativity of time: Implications and challenges of data drift on long-term effective android malware detection. *Computers & Security, in press*:102835, 2022

10. A. Guerra-Manzanares, H. Bahsi, and S. Nõmm. Hybrid feature selection models for machine learning based botnet detection in iot networks. In *2019 International Conference on Cyberworlds (CW)*, pages 324–327, 2019

11. A. Guerra-Manzanares, S. Nõmm, and H. Bahsi. Towards the integration of a post-hoc interpretation step into the machine learning workflow for iot botnet detection. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 1162–1169, 2019

12. A. Guerra-Manzanares, J. Medina-Galindo, H. Bahsi, and S. Nõmm. Medbiot: Generation of an iot botnet dataset in a medium-sized iot network. In *Proceedings of the 6th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP*, pages 207–218, 2020

13. A. Guerra-Manzanares, J. Medina-Galindo, H. Bahsi, and S. Nõmm. Using medbiot dataset to build effective machine learning-based iot botnet detection systems. In *International Conference on Information Systems Security and Privacy*, pages 222–243. Springer, 2020

Elulookirjeldus

1. Isikuandmed

Nimi	Alejandro Guerra Manzanares
Sünniaeg ja -koht	19.04.1989, Suria, Barcelona, Hispaania
Kodakondsus	Hispaania

2. Kontaktandmed

Aadress	Tallinn University of Technology, School of Information Technologies, Department of Software Science, Centre for Digital Forensics and Cyber Security, Ehitajate tee 5, 19086 Tallinn, Estonia
Telefon	+34 669 12 79 63
E-post	alejandro.guerra@taltech.ee

3. Haridus

2018–...	Tallinn University of Technology, School of Information Technologies, Information and Communication Technology, PhD studies
2017–2018	Tallinn University of Technology, Faculty of Information Technologies, Cyber Security, specialty in Digital Forensics, MSc *cum laude*
2013–2017	Polytechnic University of Catalonia, EPSEM, ICT Engineering, BSc *cum laude*
2009–2013	Autonomous University of Barcelona, Faculty of Law, Criminology, BSc *cum laude*

4. Keelteoskus

hispaania keel	emakeel
katalaani keel	emakeel
inglise keel	kõrgtase
eesti keel	algtase

5. Autasud

- 2018, Estonian Research Council Master's Thesis Contest. 3rd Prize award winner. Category of Natural Sciences and Engineering.

6. Kaitstud lõputööd

- 2018, *Application of full machine learning workflow for malware detection in Android on the basis of system calls and permissions*, MSc, supervisors Prof. Hayretdin Bahsi and Dr. Sven Nõmm, Tallinn University of Technology
- 2017, *HoneyIo4 - The construction of a virtual, low-interaction IoT Honeypot*, BSc, supervisors Dr. Francisco del Aguila Lopez and Prof. Hayretdin Bahsi, Polytechnic University of Catalonia
- 2013, *Análisis documental de la interceptación de las telecomunicaciones en España*, BSc, supervisor Dr. Francesc Xavier Moreno Oliver, Autonomous University of Barcelona

7. Teadustegevus

Teadusartiklite, konverentsiteeside ja konverentsiettekannete loetelu on toodud ingliskeelse elulookirjelduse juures.

ISSN 2585-6901 (PDF)
ISBN 978-9949-83-899-8 (PDF)