TALLINN UNIVERSITY OF TECHNOLOGY

School of Information Technologies

Department of Software Science

Exploring the Complexity Criteria of Mobile Malware: An Examination of its Impact on the Analysis Process

Master’s Thesis

Author: Fathin Dosunmu

Supervisor: Dr. Hayretdin Bahşi, PhD

Center for Digital Forensics and Cyber Security
Tallinn University of Technology Tallinn, Estonia

TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia Teaduskond

Tarkvarateaduse Instituut

Mobiilse pahavara keerukuse kriteeriumide uurimine: selle mõju uurimine analüüsiprotsessile ere

Magistritöö

Author: Fathin Dosunmu

Supervisor: Dr Hayretdin Bahşi, PhD

Center for Digital Forensics and Cyber Security
Tallinn University of Technology Tallinn, Estonia

Author's Declaration of Originality

I solemnly affirm that I am the exclusive author of this thesis. All materials utilized and references to literature and others' work have been appropriately cited. This thesis has not been submitted for evaluation elsewhere.

Author: Fathin Dosunmu
14.06.2023

Abstract

This thesis explores the intricacies and difficulties associated with mobile malware, particularly from the perspective of malware analysts. As mobile devices become more widespread, there is a corresponding rise in the complexity and sophistication of malware targeting them. The research offers an in-depth look at the factors that make mobile malware complex, focusing on how these factors impact the analysis process, the expertise needed for practical analysis, and the duration required to conduct such analyses.

This investigation adopted a combined quantitative and qualitative research design for a more holistic understanding. The methodology included a detailed review of relevant literature, creating a tool prototype, and analysing 158 samples of mobile malware. The literature review aimed to identify everyday complexities in mobile malware, while the prototype tool facilitated the automated analysis and complexity scoring of each malware sample.

Findings from the study reveal that the complexity of mobile malware varies considerably, depending on factors such as the malware's design, the obfuscation techniques used, the malware's behaviour, and the execution strategies employed. The study confirmed that more complex malware strains typically require more sophisticated knowledge and longer analysis times, straining resources and making effective mitigation more challenging. The study also confirmed that the complexity score of malware samples calculated in this research does increase over time.

Furthermore, the research highlighted that modern mobile malware often uses advanced obfuscation and evasion techniques, significantly contributing to its complexity. These techniques complicate detection and analysis, requiring analysts to adapt their approaches and constantly learn new skills. The findings suggest that a deeper understanding of these techniques and continuous upskilling are crucial for effective and timely malware analysis.

This thesis enhances mobile malware complexity knowledge, providing valuable insights for academic researchers and cybersecurity practitioners. The findings can assist in improving mobile malware analysis techniques and developing more effective cyber defence strategies. The

research also identifies potential areas for further investigation, such as developing advanced tools and methodologies to assist in analysing and mitigating high-complexity mobile malware.

List of Abbreviations

MobileApp-in-the-Middle (MAitM).

Table of Contents

1. Introduction *10*

1.1 Background of the Study **10**

1.1.1 Relevance 10

1.1.2 Current State of the field 10

1.2 Problem Statement **13**

1.3 Research Questions **15**

1.4 Scope and Limitations of the Study **15**

2. Literature Review *16*

2.1 Complexity in Mobile Malware **17**

2.2 Existing Techniques for Malware Analysis **20**

2.3.1 Static Analysis 20

2.3.2 limitations of static analysis 22

2.3.3 Dynamic Analysis 23

2.3.4 limitations of dynamic analysis 24

2.3.5 Hybrid Analysis 24

3. Methodology *25*

3.1 Selection of Malware Samples **25**

3.2 Rationale Behind Factors influencing analysis of malware. **28**

3.3 Development of the Analysis Tool **33**

3.3.1 Analytical Model 34

3.3.2 Feature Calculation and Analysis in the script 36

3.3.3 Complexity Score Calculation 38

4. Discussion and Results *38*

4.1 Correlation Between Complexity Score and Analysis Time **43**

4.2 Correlation Between Complexity Score and APK Entropy **44**

4.3 Correlation Between Complexity Score and Code Length **46**

4.4 Correlation Between Complexity Score and File Size **47**

4.5 Correlation Between Complexity Score and Obfuscated String Count................................49

4.6 Correlation Between Complexity Score and Permissions Count...50

4.7 Evolution of Complexity Over Time 52

5. Conclusion and Future Work.. 55

References 57

Appendix 1 – Malware Sample Families 66

Appendix 2 – Scrape.py [Python Script To Fetch Benign Samples]....................................... 69

List of Tables

Table 1 Complexity Score Calculation Factors and Assigned Weights

List of Figures

Figure 1 Distribution of Complexity of Benign Samples

Figure 2 Distribution of Complexity of Malware Samples

Figure 3 Complexity Score Distribution of Malware Families

Figure 4 Scatter Plot of Complexity Score vs Analysis Time

Figure 5 Scatter Plot of Complexity Score vs APK Entropy

Figure 6 Scatter Plot of Complexity Score vs code length

Figure 7 Scatter Plot of Complexity Score vs File Size

Figure 8 Scatter Plot of Complexity Score vs Obfuscated String Count

Figure 9 Scatter Plot of Complexity Score vs Permissions Count

Figure 10 Complexity score over time

Figure 11 Box plot of complexity score by year for malware samples

List of Equations

Equation 1 Malware Analysis Complexity (MAC) Formula

Equation 2 Feature Value Normalization (FVN) Formula

1. Introduction

1.1 Background of the Study

1.1.1 Relevance

In the last decade, the use of smartphones has accelerated globally, outstripping desktop and tablet computers in terms of units shipped and usage [1]. They support various tasks, from recording videos to performing financial transactions, gaming, and networking. Equipped predominantly with either Android or iOS operating systems, with Android maintaining a leading 70.8% market share [2], these devices host millions of apps designed for various tasks [3]

However, with their increased use and versatility, smartphones store large amounts of valuable information, posing significant threats to security and privacy. Mobile malware can infect these devices to steal classified information, share and track activities, and perform unauthorised tasks. For instance, the XCodeGhost malware incident 2015 infected numerous iOS applications, leading to the unauthorised access of user information [4]. Similarly, the Judy malware 2017 infected 8.5 million and 36.5 million Android devices through the Google Play Store, generating fraudulent advertising clicks [5].
Such examples underscore the significant and ever-present threat of mobile malware. Therefore, understanding the complex criteria of these malware and their influence on the analysis process is critical to devising robust detection and mitigation strategies. This significance leads us to examine the field's current state and identify where our research can contribute.

1.1.2 Current State of the field

Mobile malware analysis has witnessed substantial progress, yet it faces increasing challenges due to the growing complexity of malware threats. The field has adopted many methods to analyse and counter these threats effectively. Static and dynamic analyses remain the foundation of mobile malware examination [6]. Static analysis dissects the code without executing it,

scrutinising the structure and contents of the application's package for potential malware. Conversely, dynamic analysis runs the application in a controlled setting and observes its behaviour to detect malicious activities.

Machine learning models have progressively been utilised in mobile malware detection, leveraging characteristics such as requested permissions, API calls, and code patterns to differentiate between harmless and malicious apps [7][8]. Meanwhile, the emergence of more advanced threats has shifted towards behaviour-based analysis, prioritising malware detection through their behaviour patterns instead of depending solely on signatures. Manual heuristics analysis is an effective strategy for malware detection. Machine learning methods are considered for recognising and identifying behaviour-based malware detection, potentially offering the best solution. The activity of each piece of malware in a simulated environment would be constantly monitored, with behavioural assessments being generated. Consequently, a significant challenge is the real-time detection of malware using signature-matching algorithms [9].

The era of big data and artificial intelligence has also seen the adoption of deep learning techniques in malware detection, aiming to unearth subtle, complex patterns in extensive datasets [10][11][14]. The inception of federated learning has opened new avenues for privacy-preserving and distributed machine learning, particularly pertinent when handling sensitive user data [12]. We have also seen the adoption of MobileApp-in-the-middle Attacks, MAitM, has demonstrated various methods to effectively bypass numerous ways of multifactor authentication [20].

Despite these advancements, the rise of AI-driven mobile malware, which leverages AI techniques to bypass detection, poses a significant challenge to current detection mechanisms [13]. In the constant cat-and-mouse game between security professionals and malicious actors, the latter continually refine their methods to evade the most sophisticated defences. The field's current state reflects a landscape of ongoing innovation and persistent challenges. The intersection of AI, deep learning, and privacy-preserving techniques promises to shape the future of mobile malware analysis. However, the escalating complexity of mobile malware necessitates continuous research and adaptation of new methodologies.

However, a detailed investigation into the complexity criteria of mobile malware and its influence on the analysis process is largely overlooked in the current body of literature. This research gap is a critical area to explore, as it may hold the key to improving existing analysis techniques and paving the way for more effective detection and mitigation strategies in the face of increasingly complex mobile malware threats.

1.1.3 Gap in the Literature

While significant strides have been made in the field of mobile malware analysis, some notable gaps and questions still need addressing. A critical area that is largely overlooked in the existing body of literature is a comprehensive understanding of the complex criteria of mobile malware and the implications these criteria have on the analysis process.

Many studies have focused on detection methodologies and mitigation strategies, often using machine learning and artificial intelligence. However, a thorough exploration of how the varying complexity levels of malware impact the success and efficiency of these techniques has been limited. Factors such as the time it takes to analyse different types of malware, the level of sophistication of malware structures, and the expertise required to dissect them are pivotal in shaping a more complete and practical approach to mobile malware analysis.

This research seeks to bridge this gap by focusing on the complexity criteria of mobile malware, examining how they affect the malware analysis process, and how different levels of complexity may require different analysis methods. The expectations are that this study's outcomes will add value by:

1. It evaluates criteria that contribute to the complexity of the Android malware analysis.
2. Contribute to the malware analyst process of analysing malware.

This research seeks to enhance the detection, analysis, and mitigation of increasingly mobile malware threats by addressing these points.

1.2 Problem Statement

Mobile malware's ubiquity and evolving complexity pose significant challenges to cyber-security professionals, IT practitioners, and organisations. Cybercriminals are progressively focusing on mobile devices because of the vast amount of valuable personal and professional data they hold. As the sophistication and complexity of mobile malware grows, so does the difficulty and time commitment involved in their analysis, detection, and mitigation.

A crucial challenge in the field lies in the complexity of mobile malware, which can vary widely depending on the malware's features, behaviour, and obfuscation techniques. This complexity impacts the difficulty and resource intensity of the analysis process and can also influence the effectiveness of detection and mitigation strategies. As malware evolves to evade detection, it often uses polymorphism, metamorphism, encryption, and packing techniques, further complicating its analysis.

Despite the critical role of complexity in the mobile malware analysis process, there is a dearth of comprehensive studies specifically focused on understanding the complexity criteria of mobile malware. As a result, the field needs a standardised approach to quantify the complexity of mobile malware, and the effect of these complexity levels on the efficiency and success of analysis methods remains to be determined. This leads to significant implications, as different malware may require different analysis methods, and the resources required for analysis could vary widely depending on the malware's complexity.

Furthermore, the evolving nature of mobile malware and the constant Development of new obfuscation and evasion techniques necessitate a dynamic approach to malware analysis. There may need to be more than a static or one-size-fits-all approach in the face of increasingly complex threats. Hence, there is a need for an adaptable methodology that accounts for the complexity of the malware and adjusts the analysis process accordingly.

As pointed out by Almomani I, Ahmed M, and El-Shafai W in their 2022 work on Android malware analysis, several factors contribute to the complexity of analyzing malware. These

factors encompass the type of analysis being performed, whether it's static, dynamic, hybrid, or vision-based, as well as the specific components of the Android application package (APK) being examined, which can include the entire APK file, the Android manifest file, the SMALI file, or the Classes.dex file, and the nature of the Android dataset being analysed (balanced or imbalanced) [15]. Despite recognizing complexity as a significant factor in malware analysis, a detailed understanding of these criteria still needs to be improved in the field. This void in research presents a critical area of exploration as it could be vital to improving existing analysis techniques and inform the Development of more effective detection and mitigation strategies.

The study analyzed a broad spectrum of factors potentially impacting the efficiency of malware analysis, viewed from a complexity perspective. It included an examination of malware dataset properties, the processes and results involved in converting APK files, the use of convolutional neural network (CNN) methods, and the evaluation standards employed [15]. Such methodology reflects an increased acknowledgment of complexity's role in the analysis of malware. Nonetheless, there remains a gap in the establishment of methods to measure and classify these complexities.

The overarching problem, therefore, is the need for more understanding and underestimation of the complexity criteria of mobile malware and its influence on the analysis process. This research aims to explore these complexity criteria, how they impact the analysis process, and how different levels of complexity may necessitate different analysis approaches. By doing so, this research seeks to contribute to a more nuanced, practical, and resource-efficient approach to mobile malware analysis.

In summary, the key issues to be addressed in this research are:

1. Lack of comprehensive academic studies specifically focused on understanding the complexity criteria of mobile malware.
2. The field's lack of a standardized approach to quantify the complexity of mobile malware.
3. The unclear impact of different complexity levels on the efficiency and success of analysis methods.

4. The Development of a prototype to calculate the complexity score of malwares automatically.

1.3 Research Questions

To address the fundamental gap in the complexity criteria of mobile malware and how these criteria influence the work of malware analysts, this research endeavoured to answer the following questions:

1. What are the most common complexity criteria of mobile malware, and how do they influence the analysis process?
2. Can a tool be developed to operationalize the identification and quantification of complexity criteria in android mobile malware way?
3. How has the complexity of mobile malware evolved over time?

1.4 Scope and Limitations of the Study

While there is a multitude of malware targeting different platforms, the scope of this study is limited explicitly to malware affecting mobile Android devices. It delves into understanding the complexity criteria of Android mobile malware, how these criteria influence the analysis process, and how the complexity has evolved.

The complexity criteria considered in this study encompass a broad range of factors, such as the sophistication of malware structures, the level of expertise required for analysis, obfuscation and evasion techniques employed by malware, and the time it takes to analyse different types of malware. The study also examines the impact of these complexity criteria on the efficiency and effectiveness of different malware analysis techniques, including but not limited to static and dynamic analyses, machine learning, behaviour-based analysis, and deep learning techniques. Finally, the study explores the trends and future implications of the escalating complexity of mobile malware, as this may inform the Development of more effective detection and mitigation strategies.

Every research has its limitations, and this study is no exception. Given yearly emergence of new malware threats, it was only feasible to cover some Android mobile malware families in the analysis. This is discussed in Chapter 3.1 Selection of Malware Samples

While the study strives to develop a comprehensive understanding of the complexity criteria of mobile malware, it should be noted that these criteria might change as malware evolves. Consequently, the findings of this study are subject to the limitations of time, as future developments in the field of malware creation and analysis may introduce new complexity criteria or modify existing ones.

The study leverages many resources, including published literature. However, specific data may remain inaccessible due to confidentiality, proprietary restrictions, or information not being in the public domain. This limitation may affect the breadth of the analysis.

The dataset used in this study may only encompass part of the spectrum of Android malware and benign applications. Although sufficient for initial analysis, the number of samples limits the generalizability of the findings. Larger datasets could reveal more nuanced patterns. The variety of APK samples, considering their source, functionality, and intricacy, is essential for comprehensive analysis. Should the dataset mainly include specific kinds of applications or neglect certain malware families, it might distort the outcomes and affect the effectiveness of the tool in diverse practical situations.

Similarly, the creation of MalDroidAnalyzer, a tool intended to operationalise complexity analysis, introduces a practical component to the research. The tool's effectiveness will be evaluated against the malware samples selected, which may not encompass the full spectrum of Android malware. Additionally, the tool's Development is subject to constraints such as available technology, time, and resources.

2. Literature Review

Chapter 2 of this thesis provides a review of the current state of literature related to mobile malware, its complexity criteria, and the implications of this complexity on the analysis process.

It also identifies potential gaps in the existing literature that the present study seeks to address. This chapter serves as a crucial foundation for the subsequent analysis and discussions, situating the current research within the broader academic and professional context.

Mobile malware has emerged as a significant topic for concern and research over the past ten years. Notably, there was a 15% increase in mobile malware installations in the first quarter of 2021 compared to the final quarter of 2020 [15]. Regarding the categories of mobile malware, RiskTool AndroidOS was the predominant type in Q1 2021, accounting for 39.7% of all detected mobile malware. Trojan-Dropper AndroidOS and Trojan AndroidOS were also prominent, making up 21.5% and 15.8%, respectively [15].

Russia, with a 0.95% share of mobile users attacked by malware, stood out as the most attacked country in the first quarter of 2021, followed by Iran and Algeria, both at 0.9% [15]. In terms of mobile malware subcategories, banking Trojans saw a substantial increase in attacks, with a 2.8 times growth in users attacked between Q4 2020 and Q1 2021 [15]. Adware attacks have also seen a significant surge, accounting for 55% of all attacks in Q1 2021, a marked increase from 29% in the previous year [15]. As such, the topic is one of considerable relevance and urgency, and it is critical to examine and understand the ongoing conversations and research in this field.

2.1 Complexity in Mobile Malware

The complexity of mobile malware has evolved significantly over the years, from simple scripts to sophisticated, multifaceted threats. Mobile malware's complexity refers to its multifaceted nature, encompassing its structure, behaviour, obfuscation techniques, and evasion strategies. Complexity is crucial in the analysis process, affecting the time, resources, and expertise required for efficient identification and resolution. This section explores the various dimensions of complexity in mobile malware and their implications for malware analysis, concentrating on Android, the most frequently targeted mobile operating system, owing to its large user base and open-source nature [27].

The widespread adoption of Android has inadvertently positioned it as a favored target for malicious software attacks. Such malware is intricately crafted to exploit the Android Dalvik

virtual machine (DVM) and its fundamental Java libraries. The intricacy of this issue stems from the malware's utilization of various concealment and evasion tactics that complicate detection, casting even the safeguards of the Google Play Store into doubt. Malicious code authors persist in ingeniously disguising harmful applications as benign ones, leveraging system flaws and seeking access to various device functionalities [28]. This study assesses a spectrum of detection strategies, embracing static, dynamic, and those based on machine learning. It underscores the observation that current detection methods fall short in identifying new, previously unseen malware and variants that use obfuscation techniques to slip past security barriers. The challenges outlined include:

- **Techniques of Evasion and Disguise:** Malicious software employs complex evasion and disguise strategies to slip through the nets of conventional detection systems, presenting a significant challenge for these methods to effectively pinpoint [28].
- **Insufficient Depth in Analysis**: Certain analytical approaches neglect to scrutinize the mobile device's RAM for forensic traces or to delve into essential aspects such as the primary logic, exploitative binaries, and native code libraries.
- **Shortcomings of Signature Dependency**: The reliance on signature recognition in numerous antivirus solutions hinders their responsiveness to malware, which results in a reduced efficacy against disguised or altered forms of mobile malware.
- **Challenges in Identifying Advanced Variants**: Although a variety of detection methods are employed, imperfections persist, and there's a notable deficit in studies that delineate the weaknesses and strengths inherent in these methodologies.

In a parallel vein, the intriguing research by S. Venkatraman and M. Alazab titled "Use of Data Visualisation for Zero-Day Malware Detection" delves into the intricacies of mobile malware, with a particular emphasis on the identification of zero-day threats. Data visualization has been recognized as a vital aid for analysts burdened with the labor-intensive task of monitoring suspicious behaviors. These approaches not only summarize the existing visualization strategies for anomaly detection but also introduce a unique matrix-based visualization for precise malware categorization [31].

The objective of the study is the detection of disguised malware by visualizing patterns of similarity in x86 IA-32 (opcode) sequences, which are typically challenging to discern through conventional methods. The suggested technique merges static and dynamic analyses of malware with similarity matrix visualization to enhance the classification and detection of zero-day threats.

Furthermore, zero-day malware, also referred to as novel malware, camouflages its operations to avoid detection. These variants are termed 'zero-day' due to the absence of a time gap between their initial assault and their subsequent identification. Traditional methods of malware detection encounter substantial obstacles when dealing with such attacks. Malware detection approaches are primarily categorized into static and dynamic analysis. Static analysis involves examining the syntax and structure of a file, while dynamic analysis observes the file's behavior during execution. Malware creators often use transformation techniques to evade detection, including adding superfluous code, reordering subroutines, and changing code locations. Additionally, the use of packers to hide complete programs further complicates reverse engineering efforts.

Understanding the complexity of mobile malware involves considering multiple factors crucial to grasping its nature and behaviour within mobile environments. A primary factor is CPU usage, representing the computational resources the malware consumes, which can impair the performance and responsiveness of the infected mobile device [14]. Another vital factor is storage size, indicating the malware's memory space, impacting storage availability for legitimate applications and user data.

Testing time and pre-processing speed are also essential elements of complexity, denoting the time needed to analyse and process the malware. These can affect the efficiency and efficacy of malware detection and analysis processes [14]. These factors are interconnected, contributing to the overall complexity of mobile malware and necessitating refined approaches for its analysis and mitigation.

The rise of novel malware variants and the use of sophisticated evasion tactics necessitate the creation and implementation of cutting-edge analysis models and containment methods [42]. The

challenge of dealing with mobile malware is intensified by the inherent constraints and limitations associated with mobile devices.

The fundamental functional limitations of mobile phones significantly impact the propagation and behavior of mobile malware [43]. These limitations can be a double-edged sword: they restrict the reach of malware but also constrain the effectiveness of anti-malware solutions. Grasping these limitations and their implications is vital for developing strong and effective malware detection and prevention systems, which can proactively tackle malware threats and protect mobile ecosystems.

In the age of Big Data and the Internet of Things (IoT), malware analysis has become exceedingly labour-intensive and intricate. Current automatic methods often struggle to identify unknown obfuscated malware, necessitating the involvement of human experts to examine extensive data volumes. Visualisation techniques can integrate human and computer analysis processes, offering a more intuitive and interactive way to present data.

2.2 Existing Techniques for Malware Analysis

The ever-increasing complexity of mobile malware has led to significant challenges in the field of cybersecurity. This section explores the existing techniques for malware analysis, focusing on the complexity criteria of mobile malware. The discussion is structured around static, dynamic, and hybrid analysis methods. It emphasises the need for a nuanced approach to address the unique challenges posed by the complexity of mobile malware.

2.3.1 Static Analysis

Static analysis is a method that examines a program's code, structure, and properties without executing it. It is widely used in malware analysis to understand its functionality, behaviour, and potential impact. In the context of mobile malware, static analysis is particularly crucial due to the increasing complexity and evolving nature of mobile malware [32].

Static analysis in mobile malware involves several techniques, including:

- **Code Inspection**: This involves analyzing the source or binary code to identify suspicious patterns, functions, and structure [33].
- **Signature-Based Detection**: This technique utilizes predefined signatures or patterns to identify known malware [33].
- **Control Flow Analysis:** This involves analyzing the control flow graph of a program to detect anomalies or malicious patterns [33].

Recent studies have started investigating the use of deep learning models in the field of cybersecurity, leveraging their remarkable capabilities in data mining, learning, and pattern recognition. These models have proven to alleviate the workload of malware analysts, especially in managing the complexities associated with IoT malware [34].

One such approach is a cutting-edge, vision-based technique has been developed for the detection and categorization of IoT malware, drawing on the deep transfer learning paradigm. This method enhances the effectiveness of detection and classification by fine-tuning pre-existing models and employing a variety of ensemble tactics, eliminating the need to construct new training models from the beginning. It utilizes a random forest voting approach to synergize the capabilities of three distinct Convolutional Neural Networks (CNNs): ResNet18, MobileNetV2, and DenseNet161 [34].

This approach's effectiveness was evaluated using the MaleVis dataset, an open-source compilation of 14,226 images in RGB format. These images represent 25 malware types and one non-malicious category. A comparative study shows that this method outperforms existing leading solutions in both detection and classification. It has achieved a precision rate of 98.74%, a recall rate of 98.67%, a specificity of 98.79%, an F1-score of 98.70%, a Matthews correlation coefficient (MCC) of 98.65%, an overall accuracy of 98.68%, and an average processing time of 672 milliseconds per malware classification instance [34].

This method showcases the potential of deep learning models to enhance the efficiency and effectiveness of malware analysis, especially when applied to IoT devices. It underscores the necessity of accounting for malware complexity in the development and implementation of these models.

2.3.2 limitations of static analysis

A major challenge in this field is addressing highly obfuscated or encrypted malware. Such malware is crafted to avoid detection by concealing its actual intent or functions, posing a significant obstacle for static analysis techniques to effectively recognize them as malicious [35]. This limitation is particularly problematic when dealing with zero-day malware, which are new or previously unknown types of malwares that have not yet been identified by security researchers. Because static analysis relies on known patterns or signatures to detect malware, it can struggle to identify these new threats [36].

Another limitation of static analysis is the high rate of false positives it can produce. False positives occur when a benign application or piece of code is incorrectly identified as malicious. This can lead to unnecessary alerts and can require significant manual effort to resolve [37]. Moreover, static analysis demands considerable processing capability and time to thoroughly examine each app's code, which becomes especially demanding with an extensive array of apps or with particularly intricate malware. Research conducted by Sutter and Tellenbach in 2023 unveiled FirmwareDroid, a security toolkit for analyzing Android firmware, which is distributed as open-source. It streamlines the process of extracting and evaluating pre-installed software [35]. Their research underscored how resource-heavy static analysis can be. Utilizing FirmwareDroid, they scrutinized 5,728 samples of Android firmware from multiple manufacturers, which included a review of 75,141 unique pre-installed Android applications [35]. Although the analysis was exhaustive, it underscored the sheer amount of computing power and time required, reflecting the inherent challenges static analysis faces in processing a voluminous quantity of applications or particularly elaborate malware.

Ultimately, the limitations of static analysis are highlighted by its incapacity to consider dynamic behaviors. Since it examines an application's code in its static form, without execution, it fails to

identify malicious activities that manifest only during the application's operation. As a result, malware employing dynamic tactics to dodge detection, like unpacking encrypted code at runtime, often slips past static analysis [37].

Despite these drawbacks, static analysis continues to be an essential instrument in combating mobile malware. Its capability to swiftly and effectively examine vast quantities of code renders it an invaluable initial defense. Nevertheless, these limitations emphasize the necessity for supplementary analysis methods, like dynamic analysis, to achieve a more thorough approach to malware detection.

2.3.3 Dynamic Analysis

Dynamic analysis sets itself apart from static analysis by running the application and monitoring its behavior as it unfolds. This approach is adept at identifying and assessing complex malware that employs tactics such as dynamic loading, which involves runtime library loading, function address retrieval, function execution, and subsequent library release from memory.
The advantage of dynamic analysis lies in its capability to deliver an in-depth understanding of an app's conduct by tracking its interactions within a live setting. It can uncover the way different components of an app communicate, the system operations it performs, and the resources it utilizes while running [38].

In dynamic analysis, a pivotal technique employed is the ptrace (Process Trace) system call. This function enables one process to oversee and alter the state of another process, proving particularly effective for monitoring the system calls of a targeted process. This technique helps to unveil the complexities of dynamic payloads at both Java and native code levels [38].

Additionally, artificial intelligence tools such as neural networks and decision trees are utilized to identify concealed communication channels. These tools can discern the presence of surreptitious communications by learning from historical energy consumption data, thereby unmasking threats [39].There are various tools designed for the dynamic analysis of Android applications. DroidTrace is one such ptrace-based tool, capable of tracking dynamic payload

behaviors across Java and native codes. Compatible with all Android versions and executable on actual devices, DroidTrace stands out for its adaptability [38].

2.3.4 limitations of dynamic analysis

Despite its benefits, dynamic analysis is not without its drawbacks. Identifying the exact code path that activates dynamic loading poses a challenge. Additionally, dynamic analysis can demand significant time and computational resources, especially with complex malware types. For instance, Zhang et al.'s 2023 study observed that analyzing Android malware dynamically could extend to several hours for each sample, influenced by the malware's complexity and analysis depth [40]. Such time consumption becomes problematic with extensive app quantities or intricate malware cases.

Moreover, dynamic analysis might not completely uncover behaviors of malware that are designed to evade detection, such as those that only act maliciously under certain conditions or after a delay—behaviors that might go undetected in analysis.Yet, dynamic analysis remains essential for uncovering and understanding sophisticated mobile malware, providing insights into real-time behaviors that static analysis may miss

2.3.5 Hybrid Analysis

Hybrid analysis merges static and dynamic techniques, aiming for a thorough and effective malware detection method. It attempts to address the shortcomings of each approach, like static analysis's struggle with complex code and dynamic analysis's resource demands [41]**.** Hybrid techniques might use machine learning to assess features gathered from both static and dynamic analyses. For example, a system could evaluate opcode frequency histograms from static analysis alongside network traffic patterns specific to the user from dynamic analysis.

Among the tools that incorporate hybrid approaches is A5, an automated system that blends static and dynamic malware analysis. It engages with malware in novel ways to induce malicious actions and employs both virtual and actual Android environments to catch otherwise elusive behaviors. Cuckoo Droid is another hybrid tool, leveraging Cuckoo Sandbox capabilities for analyzing Android malware via static and dynamic means [42].

Hybrid systems, despite their integrative advantage, confront complexities due to the intricate nature of dynamic analysis components like virtual platforms and input emulators. Moreover, sophisticated apps may detect emulated environments and thus avoid detection, posing a limitation to hybrid analysis's effectiveness**.** Nevertheless, hybrid analysis is invaluable for detecting and analyzing complex mobile malware, capitalizing on the combined strengths of static and dynamic analysis to form a robust strategy.

3. Methodology

Initially, the research intended to incorporate insights from industry professionals through interviews. Given the challenges in securing participation from industry professionals for interviews, the research pivoted to a more hands-on approach. Direct analysis of mobile malware samples offers a tangible and empirical method to understand the complexity criteria of malware, providing a foundation for the subsequent chapters.

3.1 Selection of Malware Samples

The robust analysis of malware complexity necessitates diverse samples encompassing a wide range of obfuscation techniques, behaviours, and attack vectors. For this study, 158 Android malware samples were meticulously chosen from a well-maintained repository known as the malware database on GitHub [48]. The malware database repository is a comprehensive collection of malware samples, providing a broad spectrum of malicious software types. It includes, but is not limited to, trojans, ransomware, adware, banking malware, botnets, email worms, and spyware. Including such a variety ensures a holistic assessment of the evolving landscape of Android malware. Benign APK samples for this research were meticulously chosen and sourced from F-Droid, which is a free and open-source repository for Android apps, utilizes a script created for this task can be found at Appendix 2 – Scrape.py [Python Script To Fetch Benign Samples. These samples were then scanned using Virus Total to ensure they were not flagged as malicious by multiple antivirus engines. This method ensures the integrity and non-malicious nature of the benign APK dataset [91]. We establish a baseline of normal behaviour

and characteristics in APK files by analyzing benign applications, this is expanded upon in Chapter 4. Discussion and Results. The complete dataset is available to download [92]

Furthermore, timestamps for the malware samples were manually obtained to facilitate a chronological analysis of complexity evolution. This was achieved by leveraging the robust capabilities of the VirusTotal API, which provides historical data on file submissions. Each malware sample's unique hash, generated during the initial assessment phase, was a critical identifier in querying the VirusTotal database. The response from VirusTotal typically includes the first submission date of the sample, which is used as a proxy for the sample's discovery date. This data acquisition process was conducted systematically, respecting the API's rate limit to ensure a comprehensive dataset. Due to the limitations of manual retrieval and potential discrepancies in first submission dates, this approach does not guarantee the exact creation date of the malware. However, it provides a reasonable approximation of its emergence in the wild.

The manual process of timestamp retrieval involved structured querying of the VirusTotal database, followed by meticulous record-keeping to pair each malware sample with its corresponding discovery timestamp. These timestamps were then collated and organised to form a temporal dataset, enabling the analysis of trends in malware complexity over time.

Each malware family exhibits unique characteristics that can illuminate specific aspects of complexity. The families of malware samples can be seen in Appendix 1 – Malware Sample Families:

- Fake Inst: Characterized by its deceptive nature, often disguising itself as legitimate applications. Its complexity arises from mimicking real applications' behavior, making discernment challenging for users and detection systems [51].
- Fakenotify: Known for generating deceptive notifications to trick users. This adds complexity, requiring analysis techniques to differentiate between legitimate and malicious notification triggers [52].
- Jitfake: Employs just-in-time tactics for revealing its malicious payload, complicating static analysis processes [53].

- Simplocker: Uses strong encryption to lock user data, necessitating a cryptographic approach to complexity analysis [54].
- LockerPin: Alters device PINs to prevent user access, adding an additional layer of complexity in restoring user control [55].
- Wannalocker: Combines ransomware tactics with worm-like capabilities, increasing difficulty in containment and decryption [56].
- Nandrobox: Notorious for displaying unwanted ads and potentially enrolling devices in botnets, obfuscating malicious network traffic as benign [57].
- Plankton, SMSsniffer, Zsone: Known for abusing SMS services, complicating billing and spam detection systems. They may intercept and manipulate SMS messages, requiring complex analysis for message flow and content [58].
- CovidLockRansomware: Emerged during the COVID-19 pandemic, leveraging topical fear to extort victims, indicating an evolution in social engineering tactics [59].
- DoubleLockerRansomware: Combines banking trojan functionality with ransomware, highlighting the trend of hybridization in malware types [60].

Malware behaviours span a wide range, from data theft to system disruption. To understand the full spectrum of these behaviours, the selection process drew on cybersecurity research classifications, categorising malware based on their actions post-infiltration. This diversity ensures an analysis that reflects malware authors' various tactics and strategies. For instance, the classification system proposed by Barría et al. [64] emphasises the diverse range of evasion tactics used by malware, including sophisticated obfuscation techniques like code encryption and polymorphism. This highlights the need to consider the complexity of evasion methods in malware analysis.

Furthermore, Schofield et al. [65] demonstrate the use of convolutional neural networks to classify malware based on API call streams, revealing distinct behavioural patterns among different malware types. Additionally, Chowdhury et al. [66] explore the categorisation of malware using data mining and machine learning, further underscoring the breadth of malware behaviours and the importance of employing diverse analytical methods to capture this range.

Collectively, these studies illustrate the necessity of a multifaceted approach in analysing malware, which is central to the selection criteria of this research.

The landscape of malware threats continually evolves, with authors increasingly adopting sophisticated obfuscation and evasion tactics. These techniques are not just ancillary features but core aspects of modern malware, significantly complicating the analysis and detection processes. As highlighted by Mawgoud, Rady, and Tawfik [67], these techniques have been developed to provide high evasion rates against anti-malware systems, mainly through dynamic analysis methods. This evolution underscores the necessity of including malware samples that exhibit such advanced obfuscation techniques in any robust analysis. Furthermore, the survey by Aboaoja et al. [68] emphasises the challenges and future directions in malware detection, particularly in the face of such evasion tactics. The rapid Development of these techniques, as discussed by Liu et al. [69], especially in the context of the Android platform, demonstrates their critical role in evading traditional security measures. Lastly, the game-based framework for comparing program classifiers and evaders, as proposed by Damásio et al. [70], provides a unique perspective on the effectiveness of various obfuscation techniques. This body of academic literature collectively highlights the importance of selecting malware samples that implement these advanced techniques, as they are vital for assessing the robustness and effectiveness of current analysis tools and methodologies.

The complexity of malware is intricately linked to its code structure, execution flow, and interaction with the operating system. Recognizing this multifaceted nature, the selection of malware samples for this study was strategically guided by a proprietary assessment model that is inspired by and aligned with current research in the field which is discussed further in chapter 3.2 Rationale Behind Factors influencing analysis of malware.. This model specifically focuses on evaluating key complexity indicators: code entropy, API call patterns, and permission usage.

3.2 Rationale Behind Factors influencing analysis of malware.

Permissions and API calls

These are a cornerstone of Android security, dictating what an application can access or perform on a device. The number of permissions requested by an APK can indicate potential overreach

for malicious intent, making it a primary factor in complexity scoring. Excessive permissions may signal an application's capability to conduct activities beyond its advertised functionality, necessitating a higher weight in the complexity calculation [77]. The patterns of API calls made by malware are pivotal for detection and provide a window into the complexity of malware's behaviour and operational intricacies. Huang et al. (2022) leveraged dynamic analysis to label malicious behaviours in Windows API calls, employing a neural network-based approach to classify and detect malware, reflecting the sophisticated patterns that malware can exhibit [73]. Similarly, Daeef et al. (2022) emphasised the efficacy of feature engineering based on API call patterns, which proves instrumental in classifying malware into families, underscoring the diverse and complex behaviours that different malware families can exhibit [74].

The complexity of malware, in this context, is derived from the depth and sophistication of its operational behaviours as manifested through API calls. Complex malware tends to use API calls in non-standard sequences or to access unusual combinations of system resources, which can be a strong indicator of malicious intent and sophistication. This is particularly relevant in the case of advanced persistent threats (APTs) and multi-stage malware, which may use API calls to stealthily establish persistence, exfiltrate data, or perform other malicious activities over an extended period.

The complexity indicators derived from API call patterns are crucial in our study. They provide insight into the sophistication level of the malware's behaviour, which might not be apparent through static analysis alone. The MalDroidAnalyzer tool assesses these patterns within the static context of Android APKs, considering the permissions that govern API access and the potential for their abuse. The tool's analysis includes examining the API call structures embedded within the code and the permissions that could facilitate such calls, contributing to a complexity score that reflects both the potential for and the sophistication of malicious behaviour.

Thus, while Huang et al. and Daeef et al. focus on the detection capabilities provided by analysing API calls, our study extends this by using such analyses as a measure of complexity, integrating it into a broader assessment framework that captures the multifaceted nature of Android malware threats.

Native Code

Using native code libraries in Android applications can serve dual purposes: while they may enhance performance, they may also be employed to obfuscate malicious behaviour. Native code execution, due to its direct interaction with the system at a lower level, presents considerable challenges in terms of monitoring and analysis compared to its Java counterparts. It can bypass several of the security mechanisms provided by the Android runtime environment, thus necessitating its consideration as a significant factor in the complexity analysis of an application.

In the MalDroidAnalyzer tool, we quantify this complexity factor by enumerating the native libraries included within the APK. This is achieved by parsing the APK file structure to list all native library files (.so files). Furthermore, the tool inspects the APK for any JNI references that indicate calls to native functions. Both the count of native libraries and the diversity of native function calls are combined to form an aggregate score that reflects the extent of native code usage.

Specifically, the tool's analysis accounts for:

- Native Library Count: Each native library used by the APK increments the complexity score, with the assumption that each library could potentially introduce additional complexity.
- Unique Native Function Calls: A higher number of distinct native function calls suggests a more intricate use of native code, which could correlate with complex or obfuscated malware behavior.

By integrating these measurements, the weight assigned to the native code complexity factor in the MalDroidAnalyzer is justified. The tool encapsulates this aspect of complexity to provide a more comprehensive understanding of the application's potential threat level, reflected in its overall complexity score [78].

Obfuscated strings

Obfuscated strings within an APK serve a nefarious purpose: to veil the true intent of the code from both human analysts and automated analysis tools. The presence of obfuscated strings is a significant indicator of complexity due to the challenges they pose in the reverse engineering process. To quantify this aspect of complexity, the MalDroidAnalyzer tool employs a heuristic-based approach, evaluating strings based on their entropy—a measure of randomness in their character distribution—and their length.

Strings with unusually high entropy values and lengths exceeding typical use cases are flagged as potentially obfuscated. The tool then counts the number of such obfuscated strings, with each one incrementing the complexity score of the APK. The rationale behind this methodology aligns with the findings of Rastogi et al. (2013), who demonstrated the effectiveness of obfuscation techniques in evading anti-malware mechanisms [79]. By considering the prevalence and sophistication of obfuscated strings, the tool's complexity scoring system reflects the degree to which an APK employs obfuscation tactics to thwart analysis and detection.

Therefore, in our complexity assessment model, obfuscated strings are weighted significantly, contributing to a composite score that encapsulates the intricacy and stealthiness of the malware. This score is essential not only for detecting the presence of malware but also for understanding the depth of its concealment strategies, offering insights into the sophistication of its design and the potential difficulties it may pose to deobfuscation efforts.

Dynamic code execution

Dynamic code execution, such as the loading of code at runtime, is a potent mechanism used by malware to evade static analysis. This technique can be used to download and execute code from external sources after installation, making the malware's behavior unpredictable and analysis more complex [80].

Entropy

APK Entropy is a measure of randomness and is used to detect encryption and obfuscation within binaries. A higher average entropy of an APK suggests sophisticated obfuscation

techniques, which increase the complexity of the analysis. Code entropy, a significant factor in determining the complexity of malware, refers to the randomness or unpredictability in the code structure of malware, posing challenges for detection and analysis. Singh and Singh (2020) illustrated that behavior-based malware detection techniques that assess runtime features, such as string features and Shannon entropy, achieve high accuracy in distinguishing between benign and malicious applications [71]. Their findings underscore the limitations of traditional static analysis methods when faced with the sophisticated obfuscation techniques increasingly employed by malware authors. Shannon entropy, therefore, becomes a critical measure of complexity, indicative of the degree to which code obfuscation can thwart signature-based detection systems.

Incorporating these insights, our study leverages a static-based approach to assess malware complexity, acknowledging that while dynamic analysis provides a clearer indication of malicious intent, static analysis offers the advantage of being executable at scale without execution of the code. The MalDroidAnalyzer tool, developed as part of this research, utilizes static analysis to calculate the Shannon entropy of code segments and string features, assessing the level of obfuscation within a malware sample. This evaluation of randomness is vital for pinpointing areas of code intentionally crafted to be cryptic, hence aiding in the generation of a complexity score for each sample.

Furthermore, the methodological relevance of entropy in analyzing malware complexity is corroborated by the work of Gibert Llauradó et al. (2018). They proposed a file-agnostic approach that leverages the visual similarity between streams of entropy to categorize malware, thereby validating the significance of entropy in the broader context of malware analysis [72]. While their approach utilizes deep learning, the MalDroidAnalyzer adapts these principles within a static analysis framework, affirming the importance of entropy as a foundational element of malware complexity assessment.

Code Length

The work of Christodorescu et al. (2005) suggests that the semantic analysis of executable code can reveal the presence of malware and its potential complexity. The longer the code, the greater

the possibility for such code to perform a variety of functions, including malicious ones. This increase in code length can correspondingly elevate the complexity of analysis, as it provides more space for malware creators to hide malicious intent within seemingly benign code structures [82]. In the context of the MalDroidAnalyzer, code length is measured as the total number of executable instructions present in the APK's bytecode. Each instruction is evaluated, contributing to a comprehensive complexity score that considers both the quantity and the potential sophistication of the code.

File Size

As noted by Crussell, Gibler, and Chen (2012), the file size of an application, especially when inflated beyond what is typical for its functionality, can be indicative of cloned applications which may include malicious additions. While a large file size is not in itself a sign of malware, it can be associated with an application that has been padded with extraneous content, which could include malicious payloads [83]. In the MalDroidAnalyzer, the APK file size is normalized to megabytes to maintain a standard scale for comparison. This normalized file size is then integrated into the complexity scoring mechanism, reflecting how the size of the application can contribute to its potential for complexity in terms of both analysis and malicious capability.

The inclusion of these factors in the MalDroidAnalyzer tool's complexity score calculation represents an effort to quantify the multifaceted nature of malware. By assessing both the code length and file size, the tool provides a more nuanced view of an APK's complexity, accounting for the varied ways in which malware may be constructed and obfuscated. These factors are weighted within the tool's scoring algorithm to reflect their relative importance as identified in the literature, ensuring that the complexity score it generates is both empirically grounded and practically informative.

3.3 Development of the Analysis Tool

The core objective of this research is not only to understand the complexity inherent in Android malware but also to develop a tool that can systematically analyse and quantify this complexity. This tool, named MalDroidAnalyzer, is designed to dissect APK files and evaluate them based on predefined complexity criteria. The development process of MalDroidAnalyzer integrates

theoretical foundations with practical considerations, ultimately aiming to provide a reliable measure of malware complexity.

MalDroidAnalyzer is the culmination of extensive research into the attributes most significantly contribute to malware complexity. The tool's design is informed by the latest findings in the field of cybersecurity. As such, the tool is built to assess various features that collectively paint a comprehensive picture of an APK's complexity, and a static-based approach of malware analysis is implemented.

The implementation of MalDroidAnalyzer leverages a Python-based environment due to its rich ecosystem of libraries and frameworks suitable for data analysis and machine learning. The tool utilises the Androguard library for APK analysis and dissection, providing access to the various metrics required for complexity scoring and it is opensource available in GitHub [93]. Each feature is extracted, normalized, and then combined into a final complexity score using a weighted formula that reflects the relative importance of each factor. The process of normalizing and weighting is explained in 3.3.1 Analytical Model

3.3.1 Analytical Model

To objectively quantify the complexity of each malware sample, MalDroidAnalyzer employs a bespoke analytical model. This model integrates a set of features meticulously extracted from the APK Samples, each representing a specific aspect of the application's behavior or structure that contributes to its overall complexity. These features include:

1. Permissions
2. Native Code
3. Obfuscated String.
4. Dynamic Code Execution.
5. APK Entropy
6. Code Length
7. File Size

The complexity score for each malware sample is determined by a weighted sum of these extracted features:

$$Complexity\ Score = \sum_{i=1}^{n} w_i . f_i$$

Equation 1 Malware Analysis Complexity (MAC) Formula

where w_i represents the weight assigned to feature i and f_i is the normalized value of feature i. This formula ensures that each feature's influence on the final score is proportional to its significance in indicating complexity.

Each feature is associated with a predefined maximum value, representing the expected upper limit in the results. This maximum value is hypothesised about what constitutes an unusually high feature value within the context of Android malware.

Normalisation ensures that each feature contributes proportionately to the overall complexity score. This process involves scaling each feature to a uniform range to ensure balanced feature contribution. Each feature is scaled to a range between 0 and 1, where 0 indicates the minimum and 1 the maximum observed value. The normalisation formula is:

$$Normalized\ Feature\ Value = \frac{Actual\ Feature\ Value}{Maxiumum\ Feature\ Value}$$

Equation 2 Feature Value Normalization (FVN) Formula

The normalize function in the code embodies this formula. It takes the actual value of the feature and the predefined maximum value as inputs. The function then returns the normalized value, ensuring it does not exceed 1 even if the actual feature value surpasses the expected maximum due to an outlier or an exceptionally complex sample. Once normalized, each feature is multiplied by its assigned weight, reflecting its relative importance in determining the complexity score. The *calculate_complexity_score* function aggregates these weighted features into a final complexity score for each APK.

The table below details the features considered and their corresponding weights based on data of existing literature as discussed in chapter 3.2 Rationale Behind Factors influencing analysis of

malware. Table 1 Complexity Score Calculation Factors and Assigned Weights shows each of the features and its corresponding weights.

Feature	Description	Assigned Weight	Weight variable
Permissions	The number of Permissions requested by the APK	1.0	W1
Native Code	The use of native code libraries within the APK	1.5	W2
Obfuscated Strings	The number of strings with high entropy	2.0	W3
Dynamic Code Execution	The use of dynamic code loading mechanisms	2.5	W4
APK Entropy	The average entropy of the APK, indicating randomness	3.0	W5
Code Length	The total length of executable code within the APK	1.0	W6
File Size	The size of the APK file normalized to megabytes	0.5	W7

Table 1 Complexity Score Calculation Factors and Assigned Weights

3.3.2 Feature Calculation and Analysis in the script

One of the main scripts developed in development of this tool prototype is called *complexity.py*. Each feature is calculated within the script as follows:

Permissions Count - The script extracts permissions requested by the APK using *a.get_permissions()* method after analyzing the APK with *AnalyzeAPK*. The complexity score

incorporates the number of permissions directly, as each permission is considered to potentially increase the complexity of the APK. This is described in detail in Chapter 3.2 Rationale Behind Factors influencing analysis of malware.

Native Code Count - Native code usage is identified with *a.get_libraries()*, which looks for native code libraries included in the APK. The complexity score accounts for the presence of native code by adding the count of native libraries identified.

Obfuscated Strings Count - The script detects obfuscated strings by calculating the entropy of each string using the entropy function. A string is considered obfuscated if it has a high entropy value (greater than 4.5) and is longer than 20 characters, as checked by *is_string_obfuscated*. It then counts the number of such obfuscated strings across all DEX files of the APK using *extract_obfuscation_features*.

Dynamic Code Execution Count - The script searches for dynamic code execution by looking for instances of classes associated with dynamic code loading (e.g., *DexClassLoader* and *PathClassLoader*) within the methods of the DEX files using *extract_dynamic_code_features*. A count of such instances is then added to the complexity score.

APK Entropy - The script calculates the average entropy of all strings in the DEX files with *calculate_apk_entropy*. This measure helps indicate the randomness or unpredictability within the APK, with higher values potentially signifying obfuscation or complexity.

Code Length - The total length of executable code is calculated by iterating through the methods in the DEX files and summing the length of the instructions for each method with *calculate_code_length*. This length is then normalized by dividing by 1000 to prevent it from disproportionately influencing the overall complexity score. The process of normalization has been discussed in the previous chapters.

File Size - The size of the APK file is obtained by *os.path.getsize(apk_path)* and then normalized to megabytes within the *process_apk_files* function. The file size is considered in the complexity score, with the understanding that larger files may contain more complex features.

3.3.3 Complexity Score Calculation

After the individual features are extracted and calculated, the script normalizes each feature to a scale of 0 to 1 using the normalize function. This is important to ensure that each feature contributes proportionally to the final score. The normalized values are then weighted according to the predefined weights in the weights dictionary, which reflect the relative importance or impact of each feature on the perceived complexity of the malware.

The weighted sum of these normalized features constitutes the APK's complexity score, as calculated by *calculate_complexity_score*. The final score is a single value that represents the complexity of the APK, integrating all the different aspects analyzed.

4. Discussion and Results

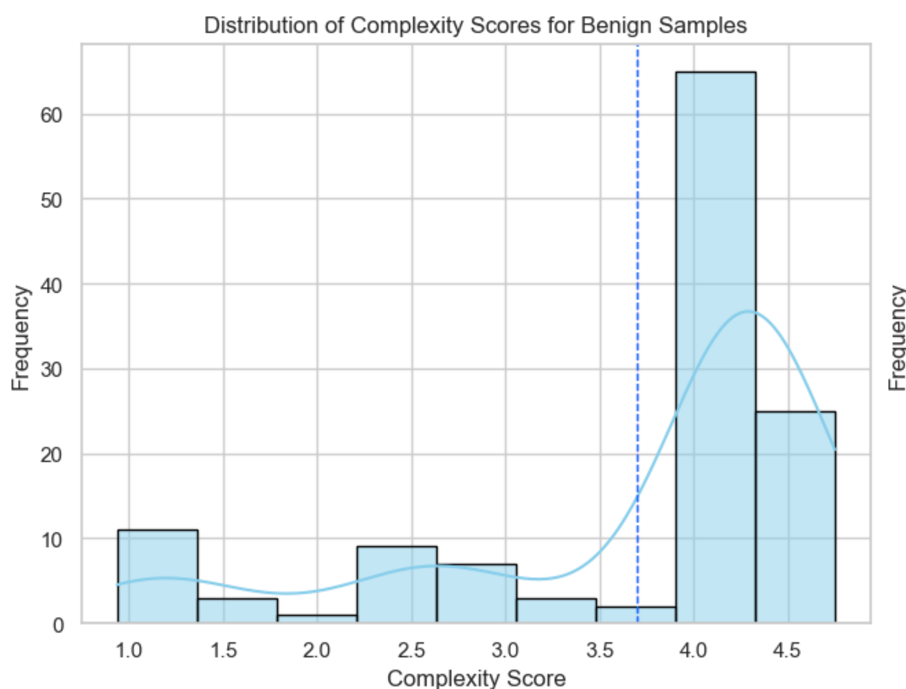

Figure 1 Distribution of Complexity of Benign Samples

In Figure 1 Distribution of Complexity of Benign Samples, the histogram is overlaid with a kernel density estimate (KDE) that represents the distribution of complexity scores for benign (non-malicious) samples. The bars represent the frequency of benign samples that fall within specific ranges of complexity scores. The x-axis shows the complexity score, while the y-axis shows the frequency of samples. Each bar's height indicates how many samples have complexity scores within the range covered by that bar. The smooth curve is the KDE, which is a way to provide a smooth representation of the distribution and is useful for visualizing the shape of the data distribution. There's a vertical dotted line that represents the mean of the complexity scores. It serves as a reference point to quickly assess the central tendency of the data. From the histogram, we can interpret several things:

- The distribution has multiple peaks (multimodal), suggesting that there are several subgroups within the benign samples that have different typical complexity scores.
- There's a frequency of samples with low complexity scores (around 1.0 to 2.0).
- The distribution seems to be right skewed, as indicated by the tail extending towards the higher complexity scores. This skewness suggests that while most benign samples have a lower complexity score, there are some with significantly higher complexity.

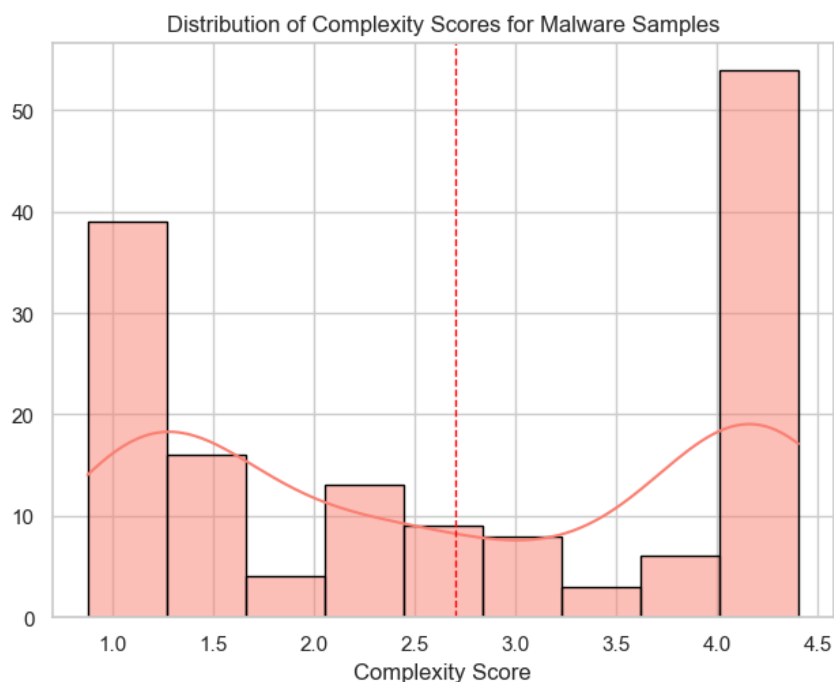

Figure 2 Distribution of Complexity of Malware Samples

Conversely, this histogram for malware samples in Figure *2* Distribution of Complexity of Malware Samples reveals:

- The distribution of malware complexity scores is also multimodal, with several peaks, suggesting variability in the complexity of malware samples.
- There's a significant peak at the higher end of the complexity scores (around 4.0 to 4.5), indicating that a considerable number of malware samples have high complexity scores.
- Unlike the benign distribution, which was right skewed, this distribution appears to have a pronounced peak at the higher end, suggesting that, within this dataset, malware tends to have a higher complexity score.

This comparison suggests that, based on this dataset, malware samples might be characterized by higher complexity scores when compared to benign samples, which could potentially be a distinguishing feature used for classification or analysis.

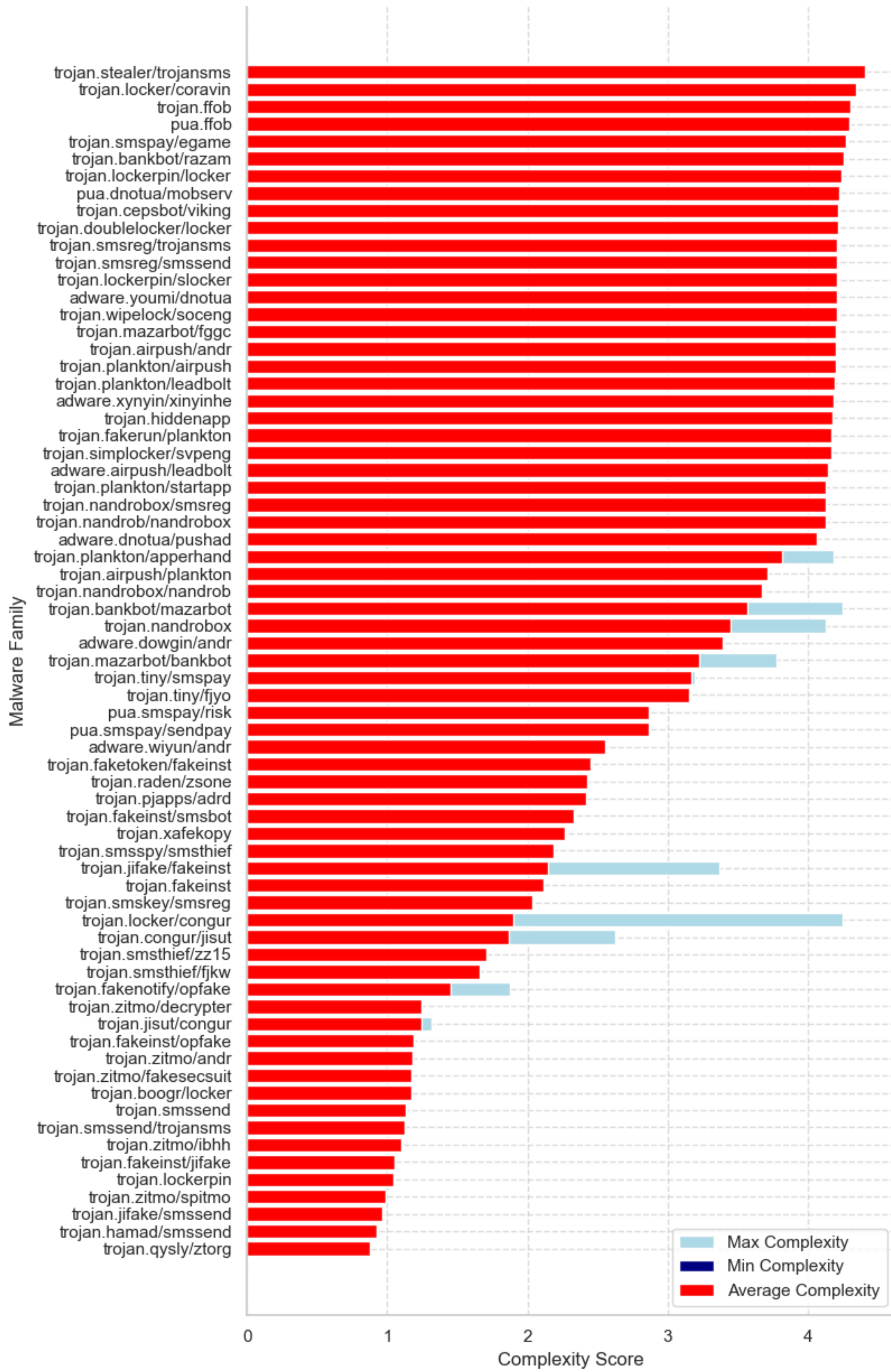

Figure 3 Complexity Score Distribution of Malware Families

To meticulously find the family of each of the malware sample used in this research, virus total API was used to extract the *suggested_threat_label* value from the API response. The list of malware families covered a total of up to 70 different families of which are listed in Appendix 1 – Malware Sample Families

In Figure *3* Complexity Score Distribution of Malware Families above, the analysis reveals significant variability in complexity scores across different malware families. The results indicate a diverse range of complexities, with some families demonstrating exceedingly high complexity levels, while others maintain lower profiles. For instance, certain families like trojan.stealer/trojansms and trojan.locker/coravin exhibit remarkably high average complexities (above 4.3), reflecting their sophisticated nature and potentially higher threat levels. Conversely, families such as trojan.jifake/smssend and trojan.hamad/smssend register lower average complexities (below 1.0), indicating less sophistication.

The distribution of complexities not only underscores the diverse tactics employed by different malware types but also highlights the evolving landscape of cyber threats. Some families show a narrow range of complexity scores, suggesting a consistent behavior pattern within that family.

The scatter plot in the following sections visualizes the relationship between the complexity score and analysis time of the APK samples, classified into two categories: benign and malware. Each point on the plot represents an individual APK sample. The different colors represent whether the APK is labeled as 'benign' or 'malware'.

4.1 Correlation Between Complexity Score and Analysis Time

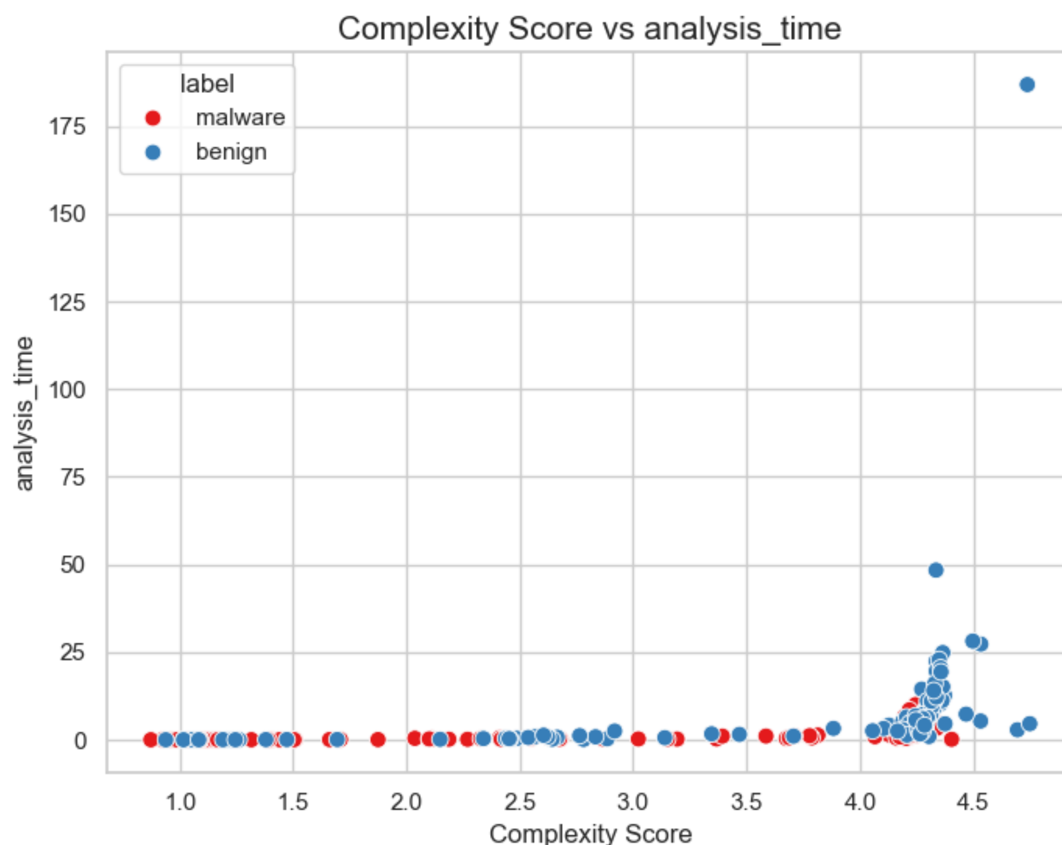

Figure 4 Scatter Plot of Complexity Score vs Analysis Time

In Figure *4* Scatter Plot of Complexity Score vs Analysis Time. Analysis Time refers to the duration required to analyze a sample by MalDroidAnalyzer, either malware or benign. This metric is calculated from the moment the analysis begins when the script is run until a conclusive result is obtained. The time is measured in seconds and the intricacies involved in deconstructing and extracting the malware features sample behavior.

The scatter plot seems to show that there is not a clear trend between the complexity score and the analysis time for either malware or benign samples. Most malware samples appear to have complexity scores distributed across the complexity score levels and the analysis time for these

samples is relatively low, mostly under 5 seconds. The benign samples are scattered across the complexity score spectrum likewise, with most of the analysis times still under 5 seconds.

However, there's a visible cluster of both benign samples and malware samples at the higher end of complexity, suggesting that a significant number of benign and malware samples are complex (at least according to the complexity score metric used). The outliers among the benign samples could indicate that a few benign applications are either very complex or take a long time to analyze, which could be due to a variety of benign reasons, such as larger size, more features, or more sophisticated coding practices. There are a few benign samples with very high analysis times (over 25 seconds, with the highest close to 175 seconds) which are outliers considering the rest of the data. One benign sample has a high complexity score (between 4.5 and 5.0) with an analysis time of around 25 seconds.

4.2 Correlation Between Complexity Score and APK Entropy

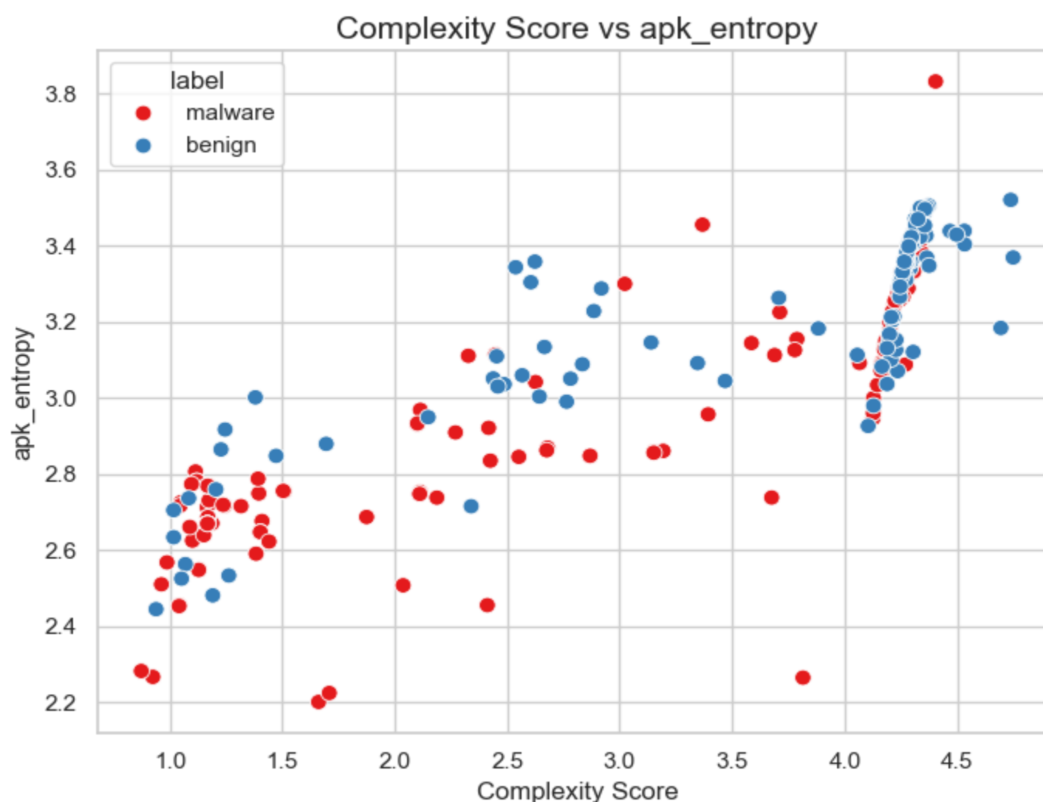

Figure 5 Scatter Plot of Complexity Score vs APK Entropy

In Figure *5* Scatter Plot of Complexity Score vs APK Entropy, the scatter plot illustrates the relationship between complexity scores and APK entropy for both malware (in red) and benign (in blue) mobile applications. It is observed that some percentage of malware samples in the dataset tend to cluster in the lower complexity score region, predominantly between scores of 1.0 and 2.5, displaying a wide range of entropy values. Likewise, some percentage of malware samples appear to also be clustered in the highest complexity region. This distribution suggests a variability in the level of obfuscation techniques employed. Notably, as the complexity score increases, there is an emergent upward trend in entropy among the malware samples, although not as pronounced as in benign applications.

Benign samples demonstrate a broader distribution across the complexity spectrum, with a concentration of points in the higher complexity score area, suggesting a greater structural or functional complexity. There is a discernible positive correlation between complexity scores and APK entropy within benign samples, indicating that as applications become more complex, the entropy tends to increase. This correlation aligns with existing literature, reinforcing the association between higher entropy and increased complexity, particularly due to obfuscation techniques.

However, it is crucial to acknowledge that the correlation depicted is influenced by the design of the complexity scoring system utilized in this study. APK entropy is a significant factor within this system, carrying the highest weight in the calculation of the complexity score. Consequently, the observed correlation is not an independent finding, but a reflection of the scoring methodology employed by the MalDroidAnalyzer tool. The dense clustering of points on the right side, particularly among benign instances, further emphasizes this methodological impact.

The visualization underscores the influence of APK entropy on the complexity score and the subsequent classification of samples as malware or benign. It confirms the relevance of entropy as a determinant in the analysis process, as posited by Singh and Singh (2020) and Gibert Llauradó et al. (2018) [71][72]. However, it also brings to the fore of highlighting the importance of considering the weightage of variables in the complexity assessment tool.

4.3 Correlation Between Complexity Score and Code Length

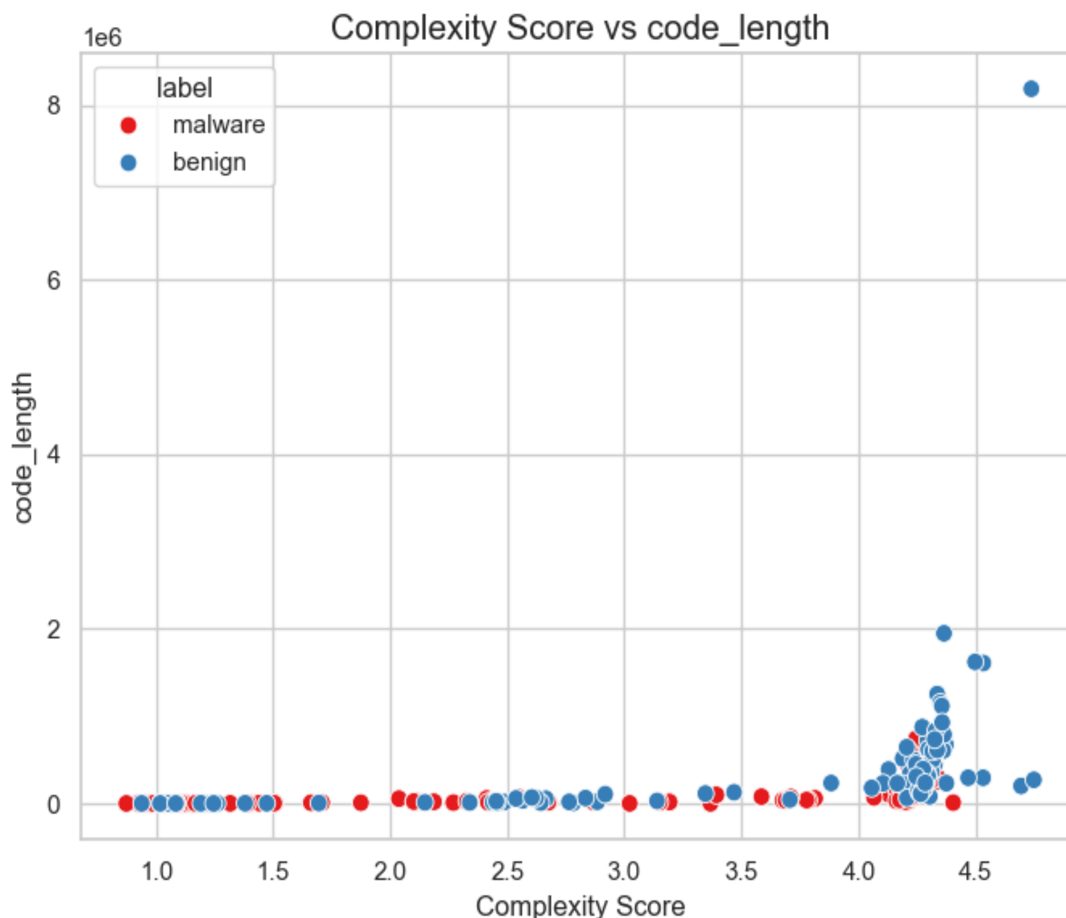

Figure 6 Scatter Plot of Complexity Score vs code length

In Figure *6* Scatter Plot of Complexity Score vs code length, the scatter plot illustrates the relationship between the complexity score of APKs and their code length. The code length for both malware and benign applications exhibits a wide range, from under 10,000 characters to over 1 million (as indicated by the Y-axis scale which extends up to 1e6, or 1,000,000 characters). A substantial number of applications, regardless of their classification as malware or benign, demonstrate a smaller codebase, characterized by code lengths of less than 50,000 characters.

Benign applications show a tendency towards higher complexity scores (above 3.5) associated with increased code lengths. Notably, outliers in the benign category present with particularly

high code lengths, including one instance approaching 1 million characters with a complexity score just below 4.5. Contrarily, most malware samples cluster at lower code lengths, though there are exceptions showcasing elevated code lengths yet maintaining lower complexity scores than comparable benign applications. This pattern suggests that while benign applications may exhibit a correlation between increasing complexity and code length, this trend is not as apparent for malware. The relative absence of a consistent upward trend in complexity with code length among malware samples could imply a strategic design by malware authors to maintain complexity without proportionally expanding code length, potentially as a tactic to evade detection.

4.4 Correlation Between Complexity Score and File Size

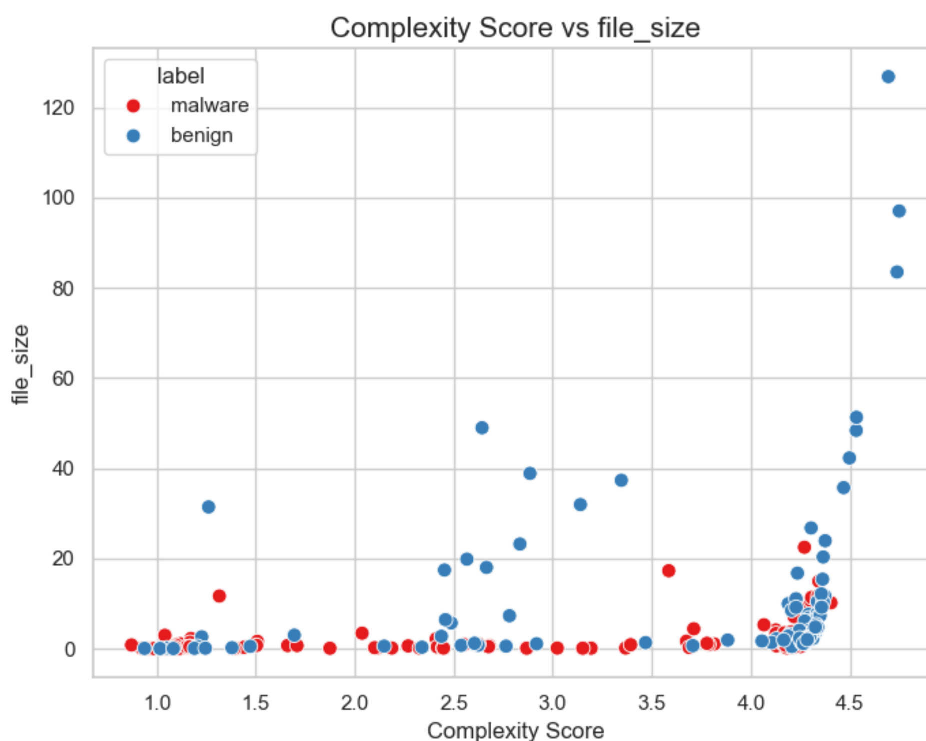

Figure 7 Scatter Plot of Complexity Score vs File Size

In Figure 7 Scatter Plot of Complexity Score vs File Size, complexity scores are plotted along the X-axis, showing a broad distribution for both benign and malware apps. There is a distinct

concentration of both app types with lower complexity scores, specifically in the range between 1.0 and 2.5.

File sizes, as shown on the Y-axis, vary widely among the APKs, extending from less than 10 MB to beyond 120 MB. Benign apps demonstrate a more diverse array of file sizes, including outliers with significantly large files. These outliers are likely reflective of applications packed with extensive features or substantial multimedia resources, contributing to their larger size.

Notably, there is a dense aggregation of data points at the higher end of the complexity scale, indicating a cluster of apps—both benign and malware—with complexity scores above 3.5. This dense region suggests that a non-negligible subset of applications, regardless of their intent, incorporate a level of complexity that is not directly proportional to file size.

While larger APKs may suggest the presence of additional features or content, ranging from innocuous to potentially malicious, an increased file size does not invariably correspond to a higher complexity score. It does, however, often coincide with a more intricate structural composition that may require more detailed scrutiny.

Echoing the insights of Crussell, Gibler, and Chen (2012) [83], an unusually large file size can be a hallmark of cloned applications that are repackaged with malicious payloads. Consequently, while file size alone is not a definitive marker for malware, it is a contributing factor in an APK’s complexity assessment. The MalDroidAnalyzer tool integrates file size into its complexity scoring system, recognizing that while it may have a subdued effect relative to other metrics, it is instrumental in delivering a comprehensive evaluation of an APK’s propensity for complexity and possible maliciousness.

4.5 Correlation Between Complexity Score and Obfuscated String Count

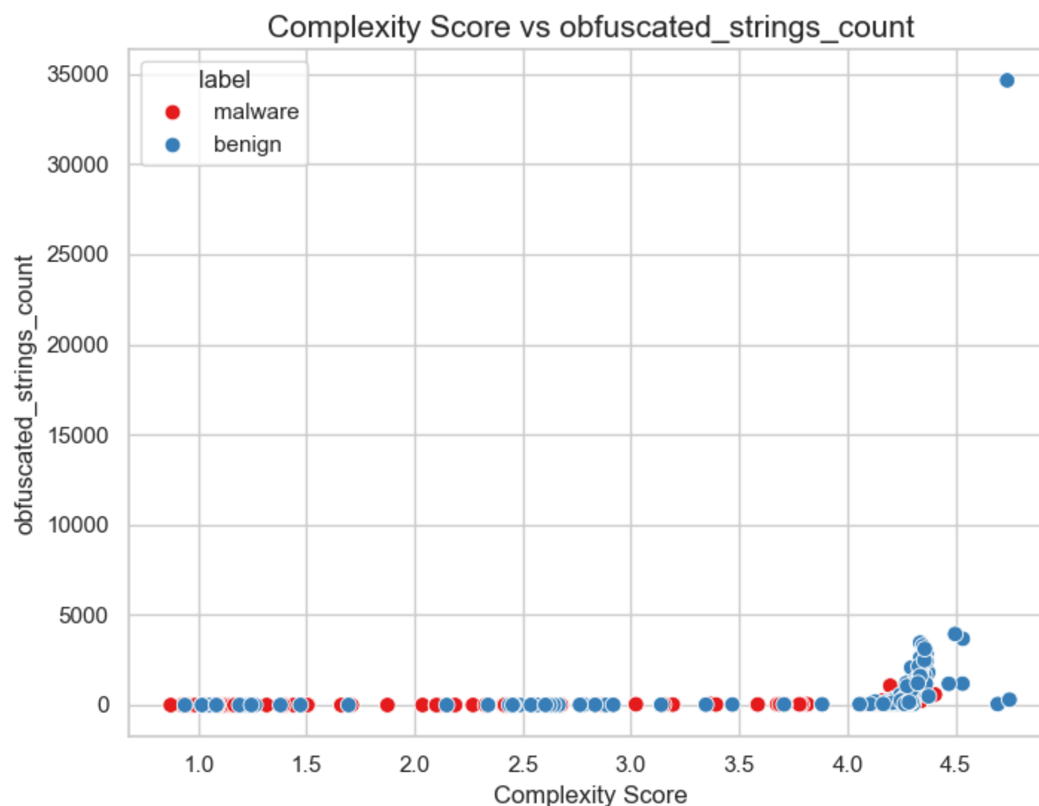

Figure 8 Scatter Plot of Complexity Score vs Obfuscated String Count

In Figure *8* Scatter Plot of Complexity Score vs Obfuscated String Count, The count of obfuscated strings among most APKs is low, with a significant number of both benign and malware applications having counts close to zero. There are a few benign applications with extremely high obfuscated strings counts, going up to 35,000, which are outliers in the dataset. Malware APKs consistently have low counts of obfuscated strings regardless of their complexity score. For benign APKs, while most also have lower counts of obfuscated strings, the presence of outliers with very high counts suggests that obfuscation is not exclusively a characteristic of malware. It might be used in benign applications for protecting intellectual property or other legitimate reasons.

4.6 Correlation Between Complexity Score and Permissions Count

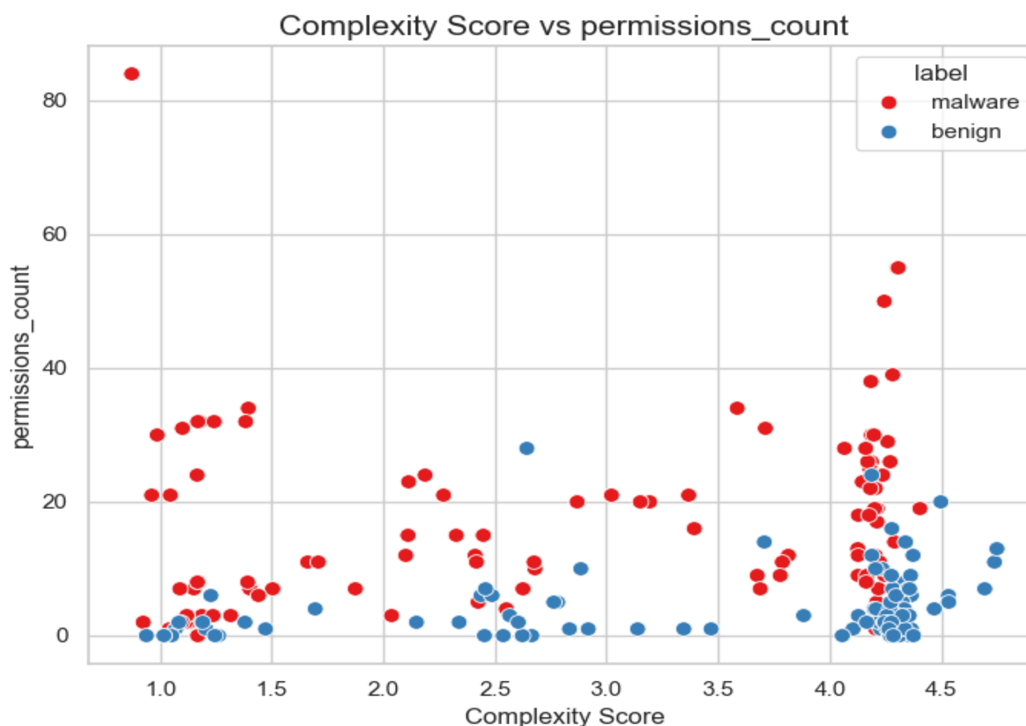

Figure 9 Scatter Plot of Complexity Score vs Permissions Count

In Figure *9* Scatter Plot of Complexity Score vs Permissions Count, the scatter plot presents the relationship between the complexity score and the permissions count for both malware and benign applications. Permissions requested range broadly from 0 to over 80 across the dataset. Notably, malware applications exhibit a wider variance in the number of permissions requested, with certain malware samples requesting a substantially high number of permissions.

While applications are distributed across a spectrum of complexity scores, there is a discernible aggregation of applications with lower complexity scores, predominantly situated between 1.0 and 3.0. Alongside this, there is also a noticeable clustering of applications—both malware and benign—at the higher end of the complexity scale, indicating that a subset of apps, irrespective of their intent, are characterized by both higher complexity and a larger number of permissions. The correlation between permissions count and complexity is nuanced. An increased number of permissions in isolation does not necessarily equate to a higher complexity score; however, it

may signal an application’s potential to overreach its expected functionalities. This overreach could manifest as an application requesting permissions that are not strictly required for its stated purpose, thereby exposing users to unnecessary risks.

In the context of malware analysis, a high permissions count, particularly when paired with other indicators of complexity, can be suggestive of malicious intent. It may indicate an application's preparedness to perform actions that could compromise user privacy or system integrity. While the permissions count does contribute to the overall complexity assessment within the MalDroidAnalyzer framework, its influence is calibrated to ensure that it does not overshadow other critical indicators of complexity and malignancy.

This balanced consideration of permissions count acknowledges its importance in the broader security landscape of Android applications. It recognizes that while a high permissions count is a noteworthy attribute, the true complexity and potential maliciousness of an APK are determined by a constellation of factors assessed in unison.

4.7 Evolution of Complexity Over Time

In this research, the earliest recorded submission date of the malware samples in the dataset is on February 9, 2011. The most recent submission date in the dataset is March 15, 2020. The average complexity score for the malware samples is approximately 2.70.

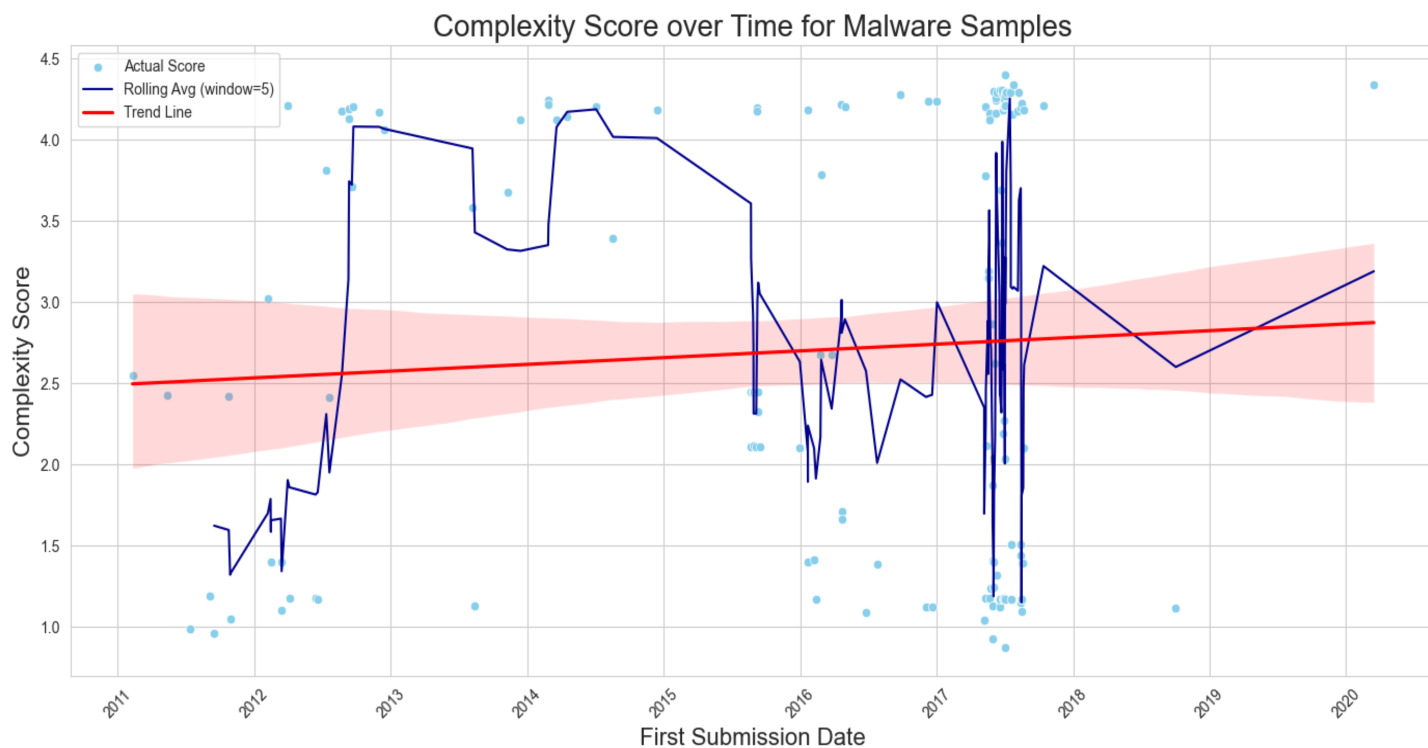

Figure 10 Complexity score over time

Figure *10* Complexity score over time represents a time series plot of complexity scores for malware samples against their first submission dates, aiming to visualize how the complexity of malware has evolved over time.

- **Actual Score (Light Blue Dots):** Each dot represents the complexity score of an individual malware sample at the time of its first submission. The spread of the dots shows the variance in complexity scores at different times.
- **Rolling Average (Dark Blue Line):** This line smooths out short-term fluctuations and shows the trend of complexity scores using a rolling average with a window of 5.

This helps to identify the general direction of the data over time, reducing the noise from individual outliers.

- **Trend Line (Red Line)**: The straight red line is likely a linear regression line that indicates the overall trend of the complexity scores. If the line has a positive slope, as it appears to have here, it suggests that there is an overall increasing trend in the complexity scores of malwares over time.
- **Confidence Interval (Red Shaded Area):** The shaded area around the trend line represents the confidence interval for the regression line. It provides a range where the true trend line is likely to exist. A wider interval suggests more uncertainty in the prediction at that point in time.

The complexity scores show considerable variance throughout the time period, with some peaks and troughs. There is an observable increase in complexity scores from around 2011 to 2013. Following that, there seems to be a period of volatility with complexity scores both increasing and decreasing. The latter years show a cluster of higher complexity scores, with some outliers showing particularly high values. The upward slope of the red trend line indicates that, despite the variability, the general trend is toward increasing complexity over time.

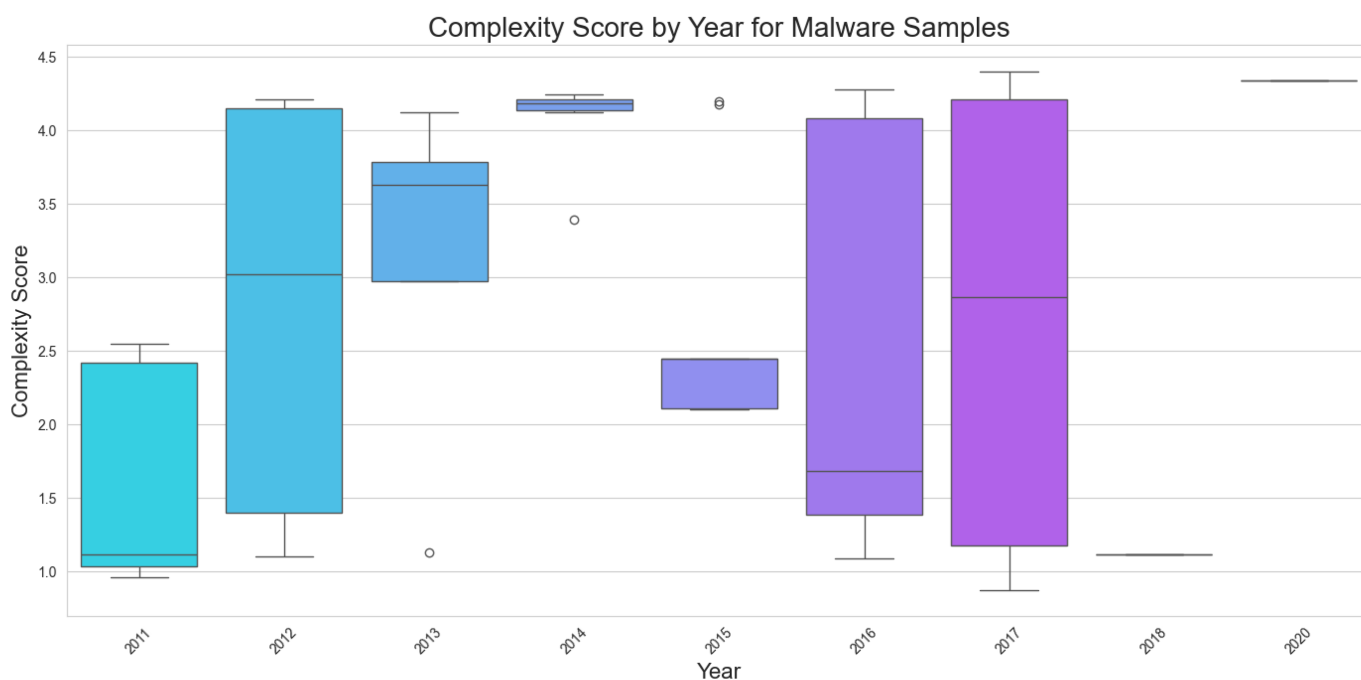

Figure 11 Box plot of complexity score by year for malware samples

Figure *11* Box plot of complexity score by year for malware samples represents the box plots of the complexity scores over time:

- **Central Rectangle (Box):** The box symbolizes the interquartile range (IQR), encompassing the central 50% of the data. The bottom and top edges of the box indicate the first quartile (25th percentile) and the third quartile (75th percentile) of the data, respectively.
- **Horizontal Line in the Box (Median)**: The line within the box denotes the median complexity score for that year. The median is the point that splits the dataset into two equal parts.
- **Whiskers**: The lines protruding from the box's top and bottom, referred to as whiskers, display the data's spread. Usually, they extend to 1.5 times the interquartile range (IQR) from both the first and third quartiles. Points that fall outside of these whiskers are often considered outliers.
- **Outliers (Circles):** The individual points that lie outside the whiskers are considered outliers. They represent complexity scores that are unusually low or high compared to the rest of the data for that year.

From Figure *11* Box plot of complexity score by year for malware samples, there is considerable variation in the complexity scores over the years. Some years have a wider range of complexity scores (e.g., 2011 and 2017), indicating more variability in malware complexity during those times.

The median complexity appears to fluctuate over the years. Without seeing the actual numbers or having a trend line, it's hard to make a definitive statement about the trend, but there does not seem to be a consistent upward or downward trend across the years. Certain years have outliers that are well above or below the main bulk of the data, suggesting the existence of malware samples with exceptionally high or low complexity scores. Some years, like 2015, have a very narrow box with no whiskers, which suggests very little variation in complexity scores among malware samples for that year. Conversely, years like 2011 and 2017 show a much wider spread, indicating a significant variation in the complexity of malware samples.

These findings indicate that over time, the complexity scores of malware samples are trending upwards, suggesting that malware is becoming more sophisticated. This increasing complexity underscores the need for continuous enhancement of security measures and malware detection systems to keep pace with the evolving threat landscape.

5. Conclusion and Future Work

This thesis has embarked on an ambitious journey to unravel the intricacies of mobile malware, within the Android ecosystem. The primary focus was to identify, analyze, quantify the factors affecting the complexity of Android mobile malware and to develop a tool to operationalize the identification and quantification of complexity criteria in a systemic way. The journey has been both challenging and enlightening, leading to several significant findings and contributions to the field of cybersecurity.

This research revealed that the complexity of mobile malware is influenced by various factors such as malware design, obfuscation techniques, behavior, execution strategies, and the evasion tactics employed. The development and successful implementation of MalDroidAnalyzer stand as a testament to the feasibility of creating tools that systematically quantify the complexity of malware. This tool utilizes a Python-based environment and a variety of analytical techniques, including Shannon entropy calculation and API call pattern analysis, to evaluate and score the complexity of Android malware. This study provides an understanding of the complexity inherent in Android malware and provides an insight into the features affecting the complexity of android malware, it offers insights into the evolving tactics of malware creators.

Reflecting on the Scope and Limitations as outlined in Chapter 1.4 Scope and Limitations of the Study, the study acknowledges certain constraints. Among these is the dataset's scope. The current dataset, while adequate to showcase MalDroidAnalyzer's functionality and to explain various complexity factors, is not without its shortcomings. A dataset of greater breadth and depth could unveil more distinct patterns and facilitate more precise outcomes. The enriched variance and comprehensive nature of a larger dataset would undoubtedly bolster the analytical prowess of the tool, enabling it to discern more nuanced behaviors within malware.

This pivot to a more substantial dataset, however, implies that MalDroidAnalyzer must be tailored to accommodate and interpret the enhanced data pool—a task that, while necessary, presents its own set of challenges. Adaptation to new data characteristics and potential recalibration of analytical algorithms may be required to maintain the tool's efficacy, representing a notable area for future enhancement.

This could tool serve as an asset for security professionals and researchers in their ongoing battle against malware threats. This study underscores the importance of continuous research and adaptation in the field of cybersecurity. The dynamic nature of malware requires that security measures and analysis tools are continually evolving. Future research should focus on enhancing the capabilities of tools like MalDroidAnalyzer, perhaps integrating machine learning and artificial intelligence to further refine the analysis and prediction of malware behavior. Also, future research should prioritize a larger and more robust dataset especially for drawing more conclusive patterns for the plots.

In conclusion, the journey through this research has not only contributed valuable insights and tools to the field of cybersecurity but also highlighted the constant need for vigilance and innovation in combating cyber threats. As malware continues to evolve in complexity and sophistication, so too must our approaches and tools for its analysis and mitigation. It is hoped that the findings and contributions of this study will serve as a steppingstone for further research and development in the ongoing effort to secure our digital world.

References

[1] MIT Technology Review, “Smartphone innovation in the third decade of the 21st century” https://www.technologyreview.com/2020/03/05/905500/smartphone-innovation-in-the-third-decade-of-the-21st-century/. March 2020

[2] Statista, Petroc Taylor “Mobile Operating Systems market Share worldwide”, https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ . July 2023

[3] Anureet Kaur, Kulwant Kaur, https://doi.org/10.1016/j.jksuci.2018.11.002 “Systematic literature review of mobile application development and testing effort estimation”. February 2022

[4] X. Gui, J. Liu, M. Chi, C. Li and Z. Lei, "Analysis of malware application based on massive network traffic," in China Communications, vol. 13, no. 8, pp. 209-221, Aug. 2016, doi: 10.1109/CC.2016.7563724.

[5] Chloe Albanesius, “'Judy' Malware Potentially Hits Up to 36.5M Android Users” https://www.pcmag.com/news/judy-malware-potentially-hits-up-to-365m-android-users. May 2017

[6] Bhat, Parnika. (2023). https://www.researchgate.net/publication/370398171_A_System_Call-based_Android_Malware_Detection_Approach_with_Homogeneous_Heterogeneous_Ensemble_Machine_Learning “A System Call-based Android Malware Detection Approach with Homogeneous & Heterogeneous Ensemble Machine Learning. Computers & Security”.

[7] Manzil, Hashida Haidros Rahima, Manohar Naik S, “Android malware category detection using a novel feature vector-based machine learning model”, https://doi.org/10.1186/s42400-023-00139-y. December 2023

[8] Laylanee M.K., Yerima S.Y., Sezer S.5. “DL-Droid: Deep learning based android malware detection using real devices (2020) Computers and Security”, https://dl.acm.org/doi/10.1016/j.cose.2019.101663. Feb 2020

[9] U. Garg, N. Sharma, M. Kumar and A. Singh, "Identification and Detection of Behavior Based Malware using Machine Learning," 2023 International Conference on Artificial

Intelligence and Smart Communication (AISC), Greater Noida, India, 2023, pp. 915-918, doi: 10.1109/AISC56616.2023.10085168.

[10] Aurangzeb, S., Aleem, M. Evaluation and classification of obfuscated Android malware through deep learning using ensemble voting mechanism. Sci Rep 13, 3093 (2023). https://doi.org/10.1038/s41598-023-30028-w

[11] Mauro Conti, Vinod P., and Alessio Vitella. 2022. Obfuscation detection in Android applications using deep learning. J. Inf. Secur. Appl. 70, C (Nov 2022). https://doi.org/10.1016/j.jisa.2022.103311

[12] D'Angelo, Gianni & Farsimadan, Eslam & Ficco, Massimo & Palmieri, Francesco & Robustelli, Antonio. (2023). Privacy-preserving malware detection in Android-based IoT devices through federated Markov chains. Future Generation Computer Systems. 148. 10.1016/j.future.2023.05.021.

[13] Daniel Gibert, Carles Mateu, and Jordi Planes. 2020. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. J. Netw. Comput. Appl. 153, C (Mar 2020). https://doi.org/10.1016/j.jnca.2019.102526

[14] Gopinath M. and Sibi Chakkaravarthy Sethuraman. 2023. A comprehensive survey on deep learning based malware detection techniques. Comput. Sci. Rev. 47, C (Feb 2023). https://doi.org/10.1016/j.cosrev.2022.100529

[15] Iman Almomani , Mohanned Ahmed, Walid El-Shafai , “Android malware analysis in a nutshell”. https://doi.org/10.1371/journal.pone.0270647. July 5, 2022

[16] Malware statistics and facts for 2023. https://www.comparitech.com/antivirus/malware-statistics-facts/

[17] Zhauniarovich, Y., Ahmad, M., Gadyatskaya, O., Crispo, B., & Massacci, F. (2015). StaDynA: Addressing the Problem of Dynamic Code Updates in the Security Analysis of Android Applications. Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, 37–48. https://doi.org/10.1145/2699026.2699111

[18] Arp, D., Spreitzenbarth, M., Huebner, M., Gascon, H., & Rieck, K. (2014). DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. Proceedings of the Network and Distributed System Security Symposium. https://doi.org/10.14722/ndss.2014.23247

[19] Almomani, I., & Gupta, B. B. (2020). Android Malware Detection Using Category-Based Machine Learning Classifiers. Journal of Information Security and Applications, 51, 102431. https://doi.org/10.1016/j.jisa.2020.102431

[20] Alcatel-Lucent. (2014). Mobile malware: a network view. Alcatel-Lucent, Motive Security Labs, 1–10.

[21] Catalano, C., Tommasi, F. Persistent MobileApp-in-the-Middle (MAitM) attack. *J Comput Virol Hack Tech* (2023). https://doi.org/10.1007/s11416-023-00484-z Canavan, J. (2005). The evolution of malicious IRC bots.

[22] Steven Furnell. "Handheld hazards: The rise of malware on mobile devices". https://doi.org/10.1016/S1361-3723(05)70210-4

[23] Zeus Virus. https://usa.kaspersky.com/resource-center/threats/zeus-virus

[24] Simplocker: First-Ever Data-Encrypting Ransomware For Android. https://www.darkreading.com/risk/simplocker-first-ever-data-encrypting-ransomware-for-android. June 2014

[25] Loapi - this Trojan us hot! https://www.darkreading.com/risk/simplocker-first-ever-data-encrypting-ransomware-for-android. June 2014

[26] Advanced Persistent Threats and Nation-State Actors https://www.cisa.gov/topics/cyber-threats-and-advisories/advanced-persistent-threats-and-nation-state-actors

[27] Global Market Share https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[28] Ashawa, Moses & Morris, Sarah. (2019). Analysis of Android Malware Detection Techniques: A Systematic Review. International Journal of Cyber-Security and Digital Forensics. 8. 177-187. 10.17781/P002605.

[29] Almomani, Iman & Ahmed, Mohanned & El-Shafai, Walid. (2022). Android malware analysis in a nutshell. PLoS ONE. 17. 1-28. 10.1371/journal.pone.0270647.

[30] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," 2012 IEEE Symposium on Security and Privacy, San Francisco, CA, USA, 2012, pp. 95-109, doi: 10.1109/SP.2012.16.

[31] Venkatraman, Sitalakshmi & Alazab, Mamoun. (2018). Use of Data Visualisation for Zero-Day Malware Detection. Security and Communication Networks. 2018. 1-13. 10.1155/2018/1728303.

[32] Damodaran, A., Troia, F.D., Visaggio, C.A. *et al.* A comparison of static, dynamic, and hybrid analysis for malware detection. *J Comput Virol Hack Tech* **13**, 1–12 (2017). https://doi.org/10.1007/s11416-015-0261-z

[33] Sihwail, Rami & Omar, Khairuddin & Zainol Ariffin, Khairul Akram. (2018). A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis. 8. 1662. 10.18517/ijaseit.8.4-2.6827.

[34] Ben Atitallah, Safa & Driss, Maha & Almomani, Iman. (2022). A Novel Detection and Multi-Classification Approach for IoT-Malware Using Random Forest Voting of Fine-Tuning Convolutional Neural Networks. Sensors. 22. 4302. 10.3390/s22114302.

[35] T. Sutter and B. Tellenbach, "FirmwareDroid: Towards Automated Static Analysis of Pre-Installed Android Apps," 2023 IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft), Melbourne, Australia, 2023, pp. 12-22, doi: 10.1109/MOBILSoft59058.2023.00009.

[36] Bilal, Mian & Rasool, Ghulam & Hashmi, Sajid & Mushtaq, Zaigham. (2023). Pragmatic Evidence on Android Malware Analysis Techniques: A Systematic Literature Review. International Journal of Innovations in Science and Technology. 1-19. 10.33411/IJIST/2023050101.

[37] Jiyun Yang, Zhibo Zhang, Heng Zhang, Jiawen Fan . Android malware detection method based on highly distinguishable static features and DenseNet, 2022. https://doi.org/10.1371/journal.pone.0276332

[38] M. Zheng, M. Sun and J. C. S. Lui, "DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability," 2014 International Wireless Communications and Mobile Computing Conference (IWCMC), Nicosia, Cyprus, 2014, pp. 128-133, doi: 10.1109/IWCMC.2014.6906344.

[39] L. Caviglione, M. Gaggero, J. -F. Lalande, W. Mazurczyk and M. Urbański, "Seeing the Unseen: Revealing Mobile Malware Hidden Communications via Energy Consumption and Artificial Intelligence," in IEEE Transactions on Information Forensics and Security, vol. 11, no. 4, pp. 799-810, April 2016, doi: 10.1109/TIFS.2015.2510825.

[40] Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X., & Zang, B. (2023). Vetting Undesirable Behaviors in Android Apps with Permission Use Analysis. ACM Transactions on Software Engineering and Methodology (TOSEM)

[41] Kouliaridis, V.; Kambourakis, G. A Comprehensive Survey on Machine Learning Techniques for Android Malware Detection. *Information* **2021**, *12*, 185. https://doi.org/10.3390/info12050185

[42] Wang, X., Yang, Y. & Zeng, Y. Accurate mobile malware detection and classification in the cloud. *SpringerPlus* **4**, 583 (2015). https://doi.org/10.1186/s40064-015-1356-1

[43] Şen, S., Aydoğan, E., & Aysan, A. (2018). Coevolution of Mobile Malware and Anti-Malware. IEEE Transactions on Information Forensics and Security, 13, 2563-2574. https://doi.org/10.1109/TIFS.2018.2824250.

[44] Yan, Q., Li, Y., Li, T., & Deng, R. (2009). Insights into Malware Detection and Prevention on Mobile Phones. , 242-249. https://doi.org/10.1007/978-3-642-10847-1_30.

[45] VirusTotal. (2022). VirusTotal - Free Online Virus, Malware and URL Scanner. Retrieved from VirusTotal

[46] Malware Zoo. Malware Zoo repositoy

[47] Contagia Mobile Malware Dump

[48] “Malware Database”, https://github.com/cryptwareapps/Malware-Database/tree/main

[49] Khoda, Mahbub E. et al. “Mobile Malware Detection - An Analysis of the Impact of Feature Categories.” *International Conference on Neural Information Processing* (2018).

[50] Ashawa, Moses and Sarah Morris. “Analysis of Mobile Malware: A Systematic Review of Evolution and Infection Strategies.” *Journal of Information Security and Cybercrimes Research* (2021): n. pag.

[51] Zhou, Y., & Jiang, X. (2012). Dissecting Android Malware: Characterization and Evolution. In IEEE Symposium on Security and Privacy. https://doi.org/10.1109/SP.2012.44

[52] Faruki, P., et al. (2015). Android Security: A Survey of Issues, Malware Penetration, and Defenses. In IEEE Communications Surveys & Tutorials. https://doi.org/10.1109/COMST.2014.2381246

[53] Tam, K., et al. (2015). CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In NDSS. https://doi.org/10.14722/ndss.2015.23243

[54] Andronio, N., Zanero, S., & Maggi, F. (2015). Heldroid: Dissecting and Detecting Mobile Ransomware. In RAID. https://doi.org/10.1145/2810103.2813629

[55] Fratantonio, Y., et al. (2016). Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop. In IEEE Symposium on Security and Privacy. https://doi.org/10.1109/SP.2016.44

[56] Continella, A., et al. (2016). ShieldFS: A Self-healing, Ransomware-aware Filesystem. In ACM CCS. https://doi.org/10.1145/2976749.2978304

[57] Zhou, W., & Zhou, X. (2012). Detecting and Analyzing Android Malware. In Proceedings of the 2012 ACM Conference on Computer and Communications Security. https://doi.org/10.1145/2382196.2382266

[58] Crussell, J., Gibler, C., & Chen, H. (2014). Attack of the Clones: Detecting Cloned Applications on Android Markets. In European Symposium on Research in Computer Security. https://doi.org/10.1007/978-3-319-11212-1_17

[59] SophosLabs. (2020). CovidLock: Analysis of a Ransomware Attack. https://doi.org/10.1007/sophoslabs.2020

[60] Štefanko, L. (2017). DoubleLocker: Innovative Android Ransomware. https://doi.org/10.1007/doublelocker.2017

[61] Cawthra, J., Ekstrom, M., Lusty, L., Sexton, J., & Sweetnam, J. (2020). Data Integrity: Identifying and Protecting Assets Against Ransomware and Other Destructive Events. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.1800-25.pdf

[62] Jang, J., & Nepal, S. (2014). A survey of emerging threats in cybersecurity. Journal of Computer System Science, 80, 973-993.

[63] Rodríguez, R., Ugarte-Pedrero, X., & Tapiador, J. (2022). Introduction to the Special Issue on Challenges and Trends in Malware Analysis. Digital Threats: Research and Practice (DTRAP), 3, 1-2."

[64] C. Barría, D. Cordero, C. Cubillos and M. Palma, "Proposed classification of malware, based on obfuscation," 2016 6th International Conference on Computers Communications and Control (ICCCC), Oradea, Romania, 2016, pp. 37-44, doi: 10.1109/ICCCC.2016.7496735.

[65] Schofield, M., Alicioğlu, G., Sun, B., Binaco, R., Turner, P., Thatcher, C., … & Breitzman, A. (2021). Comparison of malware classification methods using

convolutional neural network based on api call stream. International Journal of Network Security &Amp; Its Applications, 13(2), 1-19. https://doi.org/10.5121/ijnsa.2021.13201

[66] Chowdhury, M., Rahman, A., & Islam, R. (2017). Malware analysis and detection using data mining and machine learning classification. Advances in Intelligent Systems and Computing, 266-274. https://doi.org/10.1007/978-3-319-67071-3_33

[67] Abdelwahed, M. F., Kamal, M. M., & Sayed, S. G. (2023). Detecting malware activities with malpminer: a dynamic analysis approach. IEEE Access, 11, 84772-84784. https://doi.org/10.1109/access.2023.3266562

[68] Aboaoja, Faitouri A., Anazida Zainal, Fuad A. Ghaleb, Bander Ali Saleh Al-rimy, Taiseer Abdalla Elfadil Eisa, and Asma Abbas Hassan Elnour. 2022. "Malware Detection Issues, Challenges, and Future Directions: A Survey" *Applied Sciences* 12, no. 17: 8482. https://doi.org/10.3390/app12178482

[69] Yue Liu, Chakkrit Tantithamthavorn, Li Li, and Yepang Liu. 2022. Deep Learning for Android Malware Defenses: A Systematic Literature Review. ACM Comput. Surv. 55, 8, Article 153 (August 2023), 36 pages. https://doi.org/10.1145/3544968

[70] Thaís Damásio, Michael Canesche, Vinícius Pacheco, Marcus Botacin, Anderson Faustino da Silva, and Fernando M. Quintão Pereira. 2023. A Game-Based Framework to Compare Program Classifiers and Evaders. In Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2023). Association for Computing Machinery, New York, NY, USA, 108–121. https://doi.org/10.1145/3579990.3580012

[71] Singh, Jagsir, et al. "Detection of malicious software by analyzing the behavioral artifacts using machine learning algorithms". Information and Software Technology, vol. 121, 2020, p. 106273. https://doi.org/10.1016/j.infsof.2020.106273

[72] Gibert, Daniel, et al. "Classification of malware by using structural entropy on convolutional neural networks". Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32, no. 1, 2018. https://doi.org/10.1609/aaai.v32i1.11409

[73] Huang, Yi-Ting, et al. "Tagseq: malicious behavior discovery using dynamic analysis". Plos One, vol. 17, no. 5, 2022, p. e0263644. https://doi.org/10.1371/journal.pone.0263644

[74] Daeef, Ammar Y., et al. "Features engineering for malware family classification based api call". Computers, vol. 11, no. 11, 2022, p. 160. https://doi.org/10.3390/computers11110160

[75] Cen, Lei, et al. "A probabilistic discriminative model for android malware detection with decompiled source code". IEEE Transactions on Dependable and Secure Computing, vol. 12, no. 4, 2015, p. 400-412. https://doi.org/10.1109/tdsc.2014.2355839

[76] Xiong, Ping, et al. "Android malware detection with contrasting permission patterns". China Communications, vol. 11, no. 8, 2014, p. 1-14. https://doi.org/10.1109/cc.2014.6911083

[77] Felt, A. P., et al. (2011). Android Permissions: User Attention, Comprehension, and Behavior. In Proceedings of the Eighth Symposium on Usable Privacy and Security. https://doi.org/10.1145/2078827.2078840.

[78] Shabtai, A., et al. (2010). Andromaly: A Behavioral Malware Detection Framework for Android Devices. Journal of Intelligent Information Systems, 38, 161-190. https://doi.org/10.1007/s10844-010-0148-x.

[79] Rastogi, V., et al. (2013). Droid Chameleon: Evaluating Android Anti-malware against Transformation Attacks. In Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. https://doi.org/10.1145/2484313.2484355.

[80] Enck, W., et al. (2014). TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. ACM Transactions on Computer Systems (TOCS), 32, 1-29. https://doi.org/10.1145/2619091.

[81] Zhou, Y., & Jiang, X. (2012). Dissecting Android Malware: Characterization and Evolution. In IEEE Symposium on Security and Privacy. https://doi.org/10.1109/SP.2012.44.

[82] Christodorescu, M., et al. (2005). Semantics-Aware Malware Detection. In Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05). https://doi.org/10.1109/SP.2005.22.

[83] Crussell, J., Gibler, C., & Chen, H. (2012). Attack of the Clones: Detecting Cloned Applications on Android Markets. https://web.cs.ucdavis.edu/~hchen/paper/esorics2012.pdf

[84] Almiani, M., et al. (2020). Malware Detection in Android by Network Traffic Analysis. In Computers & Security, vol. 92. https://doi.org/10.1016/j.cose.2020.101760.

[85] Acien, A., et al. (2021). Banking Trojans and the Incorporation of Native Code as an Indicator of Malicious Intent. Journal of Computer Virology and Hacking Techniques, 17, 229–243. https://doi.org/10.1007/s11416-020-00355-5.

[86] Karbab, E. M., Debbabi, M., Derhab, A., & Mouheb, D. (2020). MalDozer: Automatic Framework for Android Malware Detection Using Deep Learning. Digital Investigation, 32, 301–320. https://doi.org/10.1016/j.diin.2020.01.004.

[87] Faruki, P., Bharmal, A., Laxmi, V., Ganmoor, V., Gaur, M. S., Conti, M., & Rajarajan, M. (2020). Android Security: A Survey of Issues, Malware Penetration, and Defenses. IEEE Communications Surveys & Tutorials, 17(2), 998-1022. https://doi.org/10.1109/COMST.2015.2392106.

[88] Sun, M., Li, M., Lui, J. C. S., & Ma, R. T. B. (2021). Monet: A User-oriented Behavior-based Malware Variants Detection System for Android. IEEE Transactions on Information Forensics and Security, 16, 1160-1175. https://doi.org/10.1109/TIFS.2020.3028218.

[89] Su, Y., et al. (2022). Deep Learning-Based Android Malware Detection with Consideration of Code Obfuscation and System API Information. Expert Systems with Applications, 186, 115739. https://doi.org/10.1016/j.eswa.2021.115739.

[90] Zhang, J., et al. (2021). A Novel Android Malware Detection Approach Using Ensemble Learning Methods. Information Sciences, 560, 181-199. https://doi.org/10.1016/j.ins.2021.02.043.

[91] https://f-droid.org/en/categories/development/

[92] https://drive.google.com/file/d/1g1I3v99CrAHgJ7tQg9L4dHSG2F3H3xue/view?usp=drive_link

[93] https://github.com/Fato07/MalDroidAnalyzer

Appendix 1 – Malware Sample Families

	Family	Number of Samples
	trojan.fakenotify/opfake	10
	trojan.congur/lockscreen	6
	trojan.faketoken/fakeinst	5
	trojan.wannalocker	5
	trojan.boogr/locker	4
	trojan.jifake/fakeinst	4
	trojan.tiny/smspay	4
	trojan.zitmo/fakesecsuit	4
	trojan.svpeng/crosate	4
	trojan.locker/congur	4
	trojan.nandrobox	3
	trojan.congur/jisut	3
	trojan.jisut/congur	3
	pua.ffob	3
	trojan.lockerpin	3
	trojan.plankton/airpush	3
	trojan.plankton/apperhand	3
	trojan.bankbot/mazarbot	2
	trojan.jisut/lockscreen	2
	trojan.svpeng/bankbot	2
	trojan.mazarbot/bankbot	2
	trojan.wannalocker/encoder	2
	trojan.plankton/leadbolt	2
	trojan.congur/locker	2
	trojan.smsreg/trojansms	2
	trojan.zitmo/decrypter	2
	trojan.simplocker/svpeng	1

	trojan.ffob	1
	trojan.hiddenapp	1
	trojan.nandrobox/nandrob	1
	trojan.nandrob/nandrobox	1
	trojan.fakeinst/opfake	1
	trojan.smssend/trojansms	1
	trojan.fakeinst/jifake	1
	trojan.hamad/smssend	1
	trojan.airpush/plankton	1
	trojan.zitmo/spitmo	1
	trojan.zitmo/ibhh	1
	trojan.zitmo/andr	1
	trojan.qysly/ztorg	1
	trojan.smspay/egame	1
	trojan.smsreg/smssend	1
	trojan.xafekopy	1
	trojan.stealer/trojansms	1
	trojan.smskey/smsreg	1
	trojan.raden/zsone	1
	pua.smspay/sendpay	1
	trojan.smssend	1
	trojan.smsspy/smsthief	1
	trojan.pjapps/adrd	1
	trojan.tiny/fjyo	1
	trojan.airpush/andr	1
	trojan.mazarbot/fggc	1
	adware.wiyun/andr	1
	trojan.svpeng	1
	trojan.bankbot/razam	1
	trojan.locker/coravin	1

	trojan.fakerun/plankton	1
	trojan.monocle/monokle	1
	adware.dnotua/pushad	1
	trojan.cepsbot/viking	1
	adware.xynyin/xinyinhe	1
	trojan.lockscreen/jisut	1
	adware.dowgin/andr	1
	adware.youmi/dnotua	1
	adware.airpush/leadbolt	1
	trojan.doublelocker/locker	1
	trojan.hqwar/bank	1
	trojan.fakeinst/svpeng	1
	trojan.smsthief/fjkw	1
	trojan.svpeng/smsspy	1
	ransomware.wannalocker	1
	trojan.lockerpin/locker	1
	trojan.lockerpin/slocker	1
	trojan.fakeinst/smsbot	1
	pua.dnotua/mobserv	1
	trojan.jifake/smssend	1
	trojan.wipelock/soceng	1
	trojan.nandrobox/smsreg	1
	trojan.plankton/startapp	1
	trojan.smsthief/zz15	1
	pua.smspay/risk	1
Benign APK		127
Total	**Unique Samples**	**279**

Appendix 2 – Scrape.py [Python Script To Fetch Benign Samples]

```
import os
import requests
from bs4 import BeautifulSoup
import time
import re

def download_apk(download_url, output_folder):
    app_name = os.path.basename(download_url)  # Extracts the file name from the URL
    response = requests.get(download_url)
    if response.status_code == 200:
        with open(os.path.join(output_folder, app_name), 'wb') as file:
            file.write(response.content)
        print(f"Downloaded {app_name}")
    else:
        print(f"Failed to download {app_name}")

def get_app_links(category_url):
    response = requests.get(category_url)
    soup = BeautifulSoup(response.text, 'html.parser')
    return [a['href'] for a in soup.find_all('a', href=True) if '/en/packages/' in a['href']]

def get_apk_download_link(app_page_url):
    full_url = f"https://f-droid.org{app_page_url}"
    response = requests.get(full_url)
    soup = BeautifulSoup(response.text, 'html.parser')

    # Use a regular expression to find the APK download link
    apk_link_pattern = re.compile(r'https://f-droid.org/repo/.+\.apk')
    apk_link = soup.find('a', href=apk_link_pattern)
    if apk_link and 'href' in apk_link.attrs:
        return apk_link['href']
    else:
        return None

def main():
    category_url = 'https://f-droid.org/en/categories/development/4/index.html'
    output_folder = 'downloaded_apks'
    os.makedirs(output_folder, exist_ok=True)

    app_links = get_app_links(category_url)

    for app_link in app_links:
        apk_download_link = get_apk_download_link(app_link)
        if apk_download_link:
```

```
            download_apk(apk_download_link, output_folder)
            time.sleep(1)  # Respectful scraping by adding delay

if __name__ == "__main__":
    main()
```