TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Mihhail Karagjaur 192898IVSB

Ransomware Detection with Machine Learning

Bachelor's thesis

Supervisor: Toomas Lepik
Master’s degree

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Mihhail Karagjaur 192898IVSB

Lunavara tuvastamine masinõppe abil

Bakalaureusetöö

Juhendaja: Toomas Lepik
Magistrikraad

Tallinn 2023

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Mihhail Karagjaur

05.01.2023

Abstract

The ransomware attacks continue to be on the rise, Windows operating system being the common target. Traditional antivirus techniques are insufficient in detection of unknown and modified variants of existing ransomware samples, they are also unable to scale with the increasing volume of malware in general. Machine learning has been used in numerous research as an alternative method to detect malicious applications, overcoming antimalware solution’s problems.

In this thesis, the suitability of an application of a simple neural network is studied, for detection of ransomware binary files, based on the extracted features. The suitability is determined by the analysis of accuracy, produced by the defined 10 feed-forward neural network models against a training set of samples. The datasets were manually collected and consist of 250 ransomware samples, 500 malware samples and 500 of benign software samples. 1 of the malware datasets is combined with the benign dataset for training of two sets of neural networks.

The results show that the simple models achieved a high accuracy score, varying from 0.88 to 0.92 between models. Time measurements have also shown the swiftness of the solution in terms of dataset preparation and model training, showing its advantage over more complex solutions.

This thesis is written in English and is 80 pages long, including 7 chapters, 17 figures and 4 tables.

List of abbreviations and terms

AI	Artificial intelligence
AUC value	Area under the curve value
C&C	Command and control
CNN	Convolutional neural network
CPU	Central processing unit
CSV	Comma-separated values
GAN	Generative adversarial network
GB	Gigabyte
GPU	Graphical processing unit
OS	Operating system
PE	Portable executable
ROC curve	A receiver operating characteristic curve
RSA	Rivest–Shamir–Adleman
ReLU activation function	Rectifier linear unit activation function
SHA256	Secure hash algorithm – 256 bit
UTC	Coordinated universal time

Table of contents

List of figures

List of tables

1 Introduction

The current chapter presents the background of the problem area, the goals, scope limitation, and the outline of the thesis.

1.1 Description of the problem

Ransomware is a type of malware which is tasked with prevention or restriction of user's access to their device, operating system, or files. There are two common types of ransomwares: locker ransomware and crypto-ransomware. Locker ransomware prevents its victims from accessing the target system by displayed lock screen. Crypto-ransomware's approach if focused on encryption of victim's key files on the target system. After successful infection and execution both forms demand a fee from their victim in exchange of restoration of access, either to the whole system or encrypted files.

Current reports show a significant rise of the ransomware attacks over the last 2 years, reaching 623 million dollars in 2021 according to Statista. The profits and total damages from ransomware attacks has reached 49.2 million dollars based on 3729 complaints received by Internet Crime Complaint Center in 2021 [1], [2].

Considering the sheer amount of malware and a ransomware subset of it that are being created, the traditional method for malware detection through signatures and hash matching is no longer sufficient to combat the increasing number of attacks [3].

Some of the disadvantages of these solutions are the inability to account for the for zero-day threats or modified versions of existing variants of ransomware, as well as poor scalability, as the creation process often requires a manual input. These downsides may lead to the cases, when a modified instance of an existing ransomware or an unknown variant may bypass present signature-based mechanisms within an anti-malware suit and get executed, leading to the data loss, loss of access to the target device, or impeding of system's operations.

Machine learning algorithms can be used as a heuristic method for ransomware detection, possibly discovering unknown or modified variants, when the matching signatures are absent, and preventing ransomware’s execution. In addition, machine learning approaches require less manual input and are more resilient against modified and unknown variants depending on the implantation. Moreover, existing research and implementations show promising results and can be effective at detecting malware in Windows OS (Operating System) [4].

1.2 Goals of the thesis

This thesis aims to develop a proof-of-concept solution, which utilizes ANN (artificial neural network), specifically in form of feed-forward-network, to detect ransomware Windows PE (portable executable) binaries and determine the efficacy of such approach to tackle the detection of ransomware binary files, targeting Windows OS. The goal of the proof-of-concept is to be able to be trained in a simple and fast form, as much research has been focused on complex solutions, which often suffer from the slow speed of creating the training dataset and consequently adjusting and updating the developed and trained solution [4].

The proof-of-concept solution is to be used to define and train an AI (Artificial Intelligence) model, and analyse its performance. The result of the model’s performance analysis will determine, if a simple neural network can be used to detect ransomware based of its static features.

The proof-of-concept could be expanded and utilized as a part of a complete malware suit or a separate production-ready solution (as a separate research or project), attempting a fast detection of the ransomware threat, if the attack involves appearance of a ransomware binary on the target system. Another use case for the solution could be the speed up of detection and filtering of ransomware binaries during forensic work, where time savings can be of high benefit.

The scope of the thesis is limited to the detection of ransomware files, to reduce the demand on the size of the dataset and due to the suitability of the solution for detection of ransomware, considering the destructive nature of ransomware, it is especially beneficial to detect before its execution.

1.3 Outline of the thesis

The thesis consists of separate chapters, discussing separate parts of the research work.

- The first chapter provides introduction to the topic, problem statement and defines the goals of the research.
- The second chapter discuses the theoretical and technical background for the machine learning and malware fields.
- The third chapter goes through the methodology of the thesis and aspects of the practical development.
- The fourth chapter goes over solution’s design.
- The fifth chapter analyses the results and output of the developed solution.
- The sixth chapter concludes the research.
- The seventh chapter provides ideas for the topic’s future developments.
- Appendices contain additional information, referenced by the thesis.

2 Theoretical background

This chapter will provide background on the machine learning and malware fields.

2.1 Malware and ransomware

Malware is a software, designed with intention to cause disruption or damage to the target system. Possible outcomes could be, a private information leak, retrieval of unauthorized access to the target systems, removal of access to the system or a part of its contents. Many types of malwares exist: backdoor malware, botnet malware, downloaders, information-stealing malware, launchers, scareware, spam-sending malware, worms, viruses and ransomware [5], [6].

Ransomware is a type of malware which threatens the victim with the removal or publication of the victim's personal data or blocking access to the data through encryption or the whole target device through a lock screen, which cannot be bypassed. When ransomware succeeds in its operation it will request the victim to pay the ransom to restore the access to the locked data or the whole system.

2.1.1 Ransomware infection vector

Ransomware infection may happen through different attack vectors. The most common vector of attack is through malicious emails, the malicious payload within the email is sent as an email attachment from emails, distributed through spam, using botnets and other compromised and infected hosts. An exploit kit is another prominent method of infection system infection. Exploit kits are a malicious type of software with an aim to scan the target system for vulnerabilities with the intent to infect it with malware. Another method of infection is drive-by downloads, in which users are attracted to malicious websites, which execute malicious code. As the result, a malicious payload is dropped onto the target system, leading to malware installation. One more prominent method of installation is the download dropper with a small footprint, its goal is to evade present antimalware systems and contact the C&C (Command and Control) centre. This method can be utilised by threat actors to split malware or ransomware into separate packages, to be processed separately in an attempt to avoid signature-based detection by antimalware solutions. If an organisation happens to be under ransomware attack, the ransomware will attempt to spread throughout the internal network with the goal to infect as many systems as possible to potentially increase ransom [7].

2.1.2 Command and control

C&C is often reached out by the installed ransomware with an aim to receive instructions. C&C centres can respond with a different set of requests with instruction to determine ransomware's behaviour. Ransomware may report gathered information about the system back to the C&C, which allows for the threat actor to adjust the attack, depending on the state of the target system. The ransomware often may reach out to the C&C for the encryption keys, to keep them secret from the target. This established channel may further be secured with more complex protocols or services, such as Tor [7], [8].

2.1.3 Encryption and extortion

Modern ransomware often uses asymmetric encryption, this allows for the ransomware to establish a secure channel to C&C, RSA (Rivest–Shamir–Adleman) is a common cryptosystem, used for public key generation. Public key encryption allows to protect the plaintext message between a server and a client to be practically inaccessible to a 3rd party. The key factor in this process is that the public key, which can only decrypt messages that were encrypted by the corresponding private key. This private key is usually held on the C&C server, which only the threat actors have access to, therefore making it impossible for the victim to retrieve. Different variants of ransomware will encrypt files in different ways, using different encryption schemes. Symmetric algorithms generate a symmetric key and encrypt files using this key (which is also responsible for the file's decryption), the advantage of which is the performance over the public key encryption. Asymmetric keys use a public key which can encrypt, but the decryption process needs the corresponding private key, which may only be stored on the malicious C&C server [7], [8].

2.2 Portable executable binary format

Windows PE (Portable Executable) format describes the structure of modern Windows program files such as “.exe”, “.dll”, and “.sys” and defines the way they store data. PE files contain x86 instructions, data such as images, text, metadata required for the program needs in order to run [3].

The PE format was originally designed to do the following [3]:

- Instruct Windows how to load a program into memory. The PE format describes which sections of a file should be loaded into memory, and where. In addition, it defines where in the program code Windows should start a program’s execution and which dynamically linked code libraries should be loaded into memory.

- Supply the running program with media (or resources), that may be used in the course of its execution. These resources can include strings of characters like the ones in GUI dialogs or console output, as well as different forms of media, such as images or videos.

- Provide security data, such as digital code signatures. Windows uses such security data to ensure that code comes from a trusted source.

The PE format contains a series of headers, followed by serieses of sections, providing the operating system with metadata on how to load the program into memory. It also includes a series of sections that contain the actual program data. Windows loads the sections into memory in a form, corresponding to where they appear on disk [3], [6], [17].

The following set of section within the PE file is considered to be the most common [6]:

- “.text” – the only section containing executable code. The rest of the section provide the supporting data for program’s operation.

- “.rdata” – the section storing the information on program’s imports and exports.

- “.data” – the section for storage of initialised global data.

- “.rsrc” – the section, containing resources such as strings and media, which are not considered to be a part of the executable itself.

2.3 Machine learning

Machine learning is field that is dedicated towards understanding and development of algorithms and methods to solve the tasks, based on the past experience (or input data). While traditional algorithms tell the computer directly what to execute, machine-learning based systems and algorithms learn how to solve the set-out task by example [3], [11].

Nowadays, machine learning is widely used. Spam filters, credit card fraud detection, image recognition, search engines, recommendation systems, data mining and handwriting recognition are some of the fields where machine learning is widely utilised.

2.3.1 Neural networks or deep learning

Neural network is a form of machine learning algorithms, defined as a network of connected computational elements. The fundamental unit of such networks is a collection of parameters called an artificial neuron, though it’s often referred to simply as a neuron or unit. The artificial neuron was inspired by human neurons, which are the nerve cells that make up our brain and central nervous system and are largely responsible for our cognitive abilities [11].

An artificial neuron receives signals, then processes them and can send a signal neurons connected to it in the following layer. The "signal" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called edges. Neurons and edges typically have a weight that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold [11].

Typically, neurons are aggregated into layers. Different layers may perform different transformations on their inputs depending on their configuration. Signals travel from the first layer (the input layer) to the last layer (the output layer), possibly after traversing the hidden (or intermediate) layers multiple times [11].

2.3.2 Supervised and unsupervised algorithms

Two of the categories in which algorithms may fall in:

- Supervised (or labelled) learning: where an outcome variable (dependent variable, y) is predicted based on a set of predictors (independent variable, x). Algorithm's task is to determine a function from labelled training data. Labelled (also known as target or outcome) data is used to train an AI model (learn from known labelled samples) to be able to determine a label (or class) for an unknown target. Such neural networks (or models) are trained based on set of features extracted from the target objects. When the model is trained, it should be able to produce a label for the unknown instance of data based on its configuration [11].

- Unsupervised learning: a form of learning, when a label for the data is unavailable and the algorithm is configured to learn relationships between target's feature themselves, rather than between the features and predefine label. This technique results in grouping or clustering of the targets based off of the learned (or spotted) patterns and associations between target's features by the trained model [11].

- Reinforcement learning: a method with which a model attempts to determine suitable solutions to set out problems. Each attempt will be rewarded with a score, depending on the suitability or efficiency of the proposed solution. After producing a multitude of variations, a variation with the best score is going to get picked. And the process continues [11].

2.3.3 Types of neural network algorithms

There is wide variety of algorithms, a subset of them is described below:

- Feed-forward neural network – the simplest variant of a neural network. The network consists of layers (or stacks) of neurons (or parameters). The neurons may be connected to all or only a subset of the neurons within the following layer. The established connection between neurons can only direct forwards, they cannot go backwards (feed into previous neurons) or form cycles. This kind of networks is often chosen as the default option to tackling an unknown problem, if a more suitable variant of neural networks is unknown [3], [11].

- Convolutional neural network – a network, containing convolutional layers, within which connection to the following neurons is limited by a windows that slides over the inputs. This structure contributes towards local feature learning.

Thanks to such property, this network was proven to be effective at image recognition and classification [3], [11].

- Generative adversarial network - the system consists of two neural networks, competing against each other with an aim to improve each own's accuracy. The generative network's task is to generate fake samples. Discriminator network's goal is to distinguish the difference between the real values and generated ones [3], [11].

2.3.4 Model overfitting and underfitting

Overfitting and underfitting are phenomena when the trained models are unable to capture a general trend within dataset after the training process. Underfitting results in models that, despite ignoring outliers, are unable to capture the trend within the target's features, resulting in poor accuracy on previously unseen samples. The overfit models does not manage to capture the general trend within the data as well, as it highly values the exceptions (or outliers) within the dataset instead, resulting in low accuracy against previously unseen data.

3 Methodology

This bachelor's thesis followed the methodology from the CRISP-DM (Cross- Industry Standard Process for Data Mining) open standard, which is often relied on in data mining or machine learning related projects [9], [10]. The standard is followed to achieve replicability of the work and to make the process more rigorous. The thesis follows the following phases, according to the standard:

- Business Understanding
- Data Understanding
- Data Preparation
- Modelling
- Evaluation
- Deployment

3.1 Business understanding phase

Business understanding phase involves understanding project's objective, definition of a data mining (machine learning) problem and a preliminary plan to achieve the objectives [9], [10].

As it is determined within the chapter 1 "Introduction" and its subchapters, the main goal of the thesis it to develop a proof-of-concept solution, which would be simple to develop and train, easy to prepare data for and is able to identify ransomware Windows binaries from benevolent software. The preliminary plan and direction of thesis were discussed during the project's development.

3.2 Data understanding phase

Data understanding phase consists of initial data collection, study of the gathered data with an aim to identify data quality deficiencies or to acquire first insights into the data to form hypotheses or determine subsets of interest [9], [10].

The samples were gathered in the suitable format of Windows binary executable under “.exe” format. The determined features for AI model training were extracted by a custom solution, preventing possible deficiencies with the data. The purpose of data acquisition is to compile a dataset for AI model training and thus determine if the set out solution is suitable for ransomware detection.

3.2.1 Ransomware sample acquisition

The ransomware samples were sourced from “MalwareBazaar” malware sample exchange [12]. Samples were filtered under the tag of "Ransomware", and the first set amount was collected in form of Windows executables in “.exe” format.

Initial ransomware dataset consisted of the first 250 samples, with the first ransomware under the “b299675e7e4654beadcaa2c38a96bf8324bbde96904ede17fdd88ebb7fdf2748” SHA256 (Secure hash algorithm – 256 bit) hash. The sample was added to the exchange on 2022-12-30 05:30 UTC (Coordinated Universal Time). The last hash within dataset is with the “77ad72f43a71412976a9ccaa8517644ebe7ff3703e5274e965ca7de612a845e7" SHA256 hash, with the date of addition - 2022-11-09 12:48:00 UTC.

Considering the development of the practical solution, a decision was taken to extend the original ransomware dataset, additional 250 samples were gathered for the total of 500. The SHA256 hash of the latest ransomware within the "extended" dataset with 500 samples total – “e9f769b1c0a4ca0a6cab2a165d5c2614d35c58b4d47b555e04bcc971d 94c5c30”, the date of its addition - 2022-09-29 12:35 UTC.

3.2.2 Benign software sample acquisition

The benign software was collected from “SourceForge” open source and business software repository [13]. Five categories were arbitrarily picked under website’s “Open Source Software” tab:

- Software Development

- Internet
- Scientific/Engineering
- System
- Database

Provided software by the website was sorted for Windows operating system and sorted by "Most Popular" feature, by default. First 100 binary executables for Windows in ".exe" were picked from each of the 5 categories, resulting in the total amount of 500 benign samples. The targets were automatically scanned by the website on the maliciousness. None of picked samples were identified as malware.

3.3 Data preparation phase

Data preparation phase – the goal is to create a final dataset from the initial raw data, through its cleaning, transformation, or extraction of subsets of data. The dataset would contain final converted values from the acquired data [9], [10].

3 main sub-datasets are defined:

- Benign dataset, consisting of extracted and converted values from the 500 benign samples collected from "SourceForge".
- Initial malicious dataset, consisting of values, collected from 250 ransomware samples.
- Extended malicious dataset, consisting of the original 250 samples from the initial dataset and additional 250, totalling 500 samples.

The datasets are stored in form of CSV (Comma-Separated Values) files, containing rows of values, collected from separate samples' PE headers. Rows of data are vectors (alternatively arrays, collections, or lists) of scalar numerical values with a value ranging from 0 to 1.

The gathering of malicious samples and their feature extraction were conducted within a virtualised instance of Ubuntu OS, version 22.04 LTS. Benign samples and their features were collected on the main machine, with Windows 10 Enterprise LTSC OS, version 1809. The main machine had “AMD Ryzen 7 3700x” processor with 32 GB (GigaByte) of memory. The virtualised instance of Ubuntu OS was configured with 4 CPU (Central Processing Unit) scores and 8 GB of memory.

During the training process of the AI model, the total combined dataset of benign samples with either initial malicious or extended ones in randomly shuffled form, is divided into the 3 subsets and is used for AI model’s training:

- 70% of the total dataset is utilised for the training of the AI model.
- 15% of the total dataset is utilised for the validation and control of the training process of the AI model.
- 15% of the total dataset is used separately, to generate predictions from the ready and finished AI model for further analysis of model’s accuracy.

The ratios were chosen according to common practices within the data science field, setting off most of the dataset for training and the smaller subset for some form of validation of the target models.

3.4 Modelling phase

During the modelling phase, the neural network machine learning or data mining solution is created, based upon the prepared dataset and variables, and its training is conducted [9], [10].

Feed-forward neural network type of neural network is chosen, as it is the simplest and first kind of neural network [3], and for the lack of knowledge of what other existing or custom type of AI model would fit the current problem better. This type of network is a suitable starting point for a proof-of-concept solution, which, if shows promise, can be further modified into a custom solution, to better suit the objective.

The model is of supervised nature, as it is a simple option to ensure AI model’s ability to classify or recognise complex targets. Supervision is also a more effective measure in tackling malware recognition [3].

Considering the absence of an existing and clear answer towards what shape of a model (a suitable number of hidden layers or total amount of neurons), 5 models are defined with different shapes, to relieve thesis results' dependency on the accuracy of one model and, in addition, to observe if the model's shape has a significant influence on model's accuracy, considering constraints in form of the dataset and features used.

All the models have an identical initial layer, consisting of 451 neurons (or units, parameters) equal to the number of features defined and extracted from PE targets. The final or outcome layer contains only one neuron (or unit) which will produce the prediction value within the range of 0 and 1, later the value is rounded, to the 0 or 1.

The 5 models defined:

- Model variant A – contains 7 hidden layers, each layer's neuron count is equal to half of count from the previous layer. The current model was inspired by an example malware detector solution from the 11th chapter "Building a neural

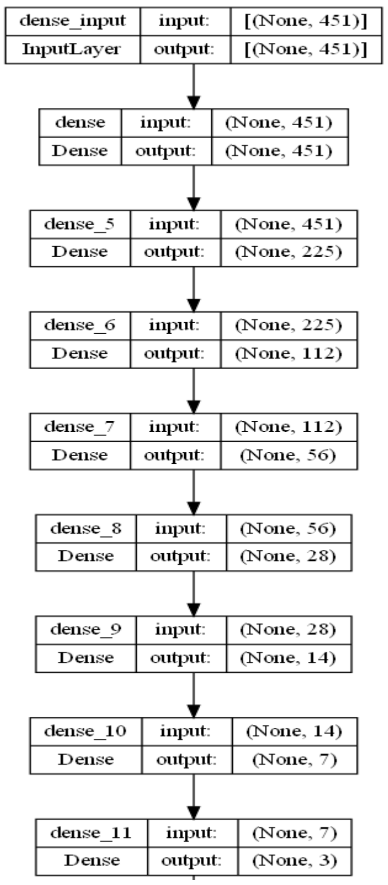

dense_input | input: | [(None, 451)]
InputLayer | output: | [(None, 451)]
dense | input: | (None, 451)
Dense | output: | (None, 451)
dense_5 | input: | (None, 451)
Dense | output: | (None, 225)
dense_6 | input: | (None, 225)
Dense | output: | (None, 112)
dense_7 | input: | (None, 112)
Dense | output: | (None, 56)
dense_8 | input: | (None, 56)
Dense | output: | (None, 28)
dense_9 | input: | (None, 28)
Dense | output: | (None, 14)
dense_10 | input: | (None, 14)
Dense | output: | (None, 7)
dense_11 | input: | (None, 7)
Dense | output: | (None, 3)

Figure 1. A graph showing model A's shape.

network malware detector with Keras" from the book "Malware Data Science" [3]. The figure 1 depicts the plot, showing the model's structure.

- Model variant B – has 1 hidden layer, containing 100 neurons. The number of layers is chosen arbitrarily to establish a certain baseline for comparison against the rest of the models. Model C-E follow similar idea, to be more comparable to the current model. The number of neurons was specifically chosen to be lower the number of neurons within the input layer, following the observation of different

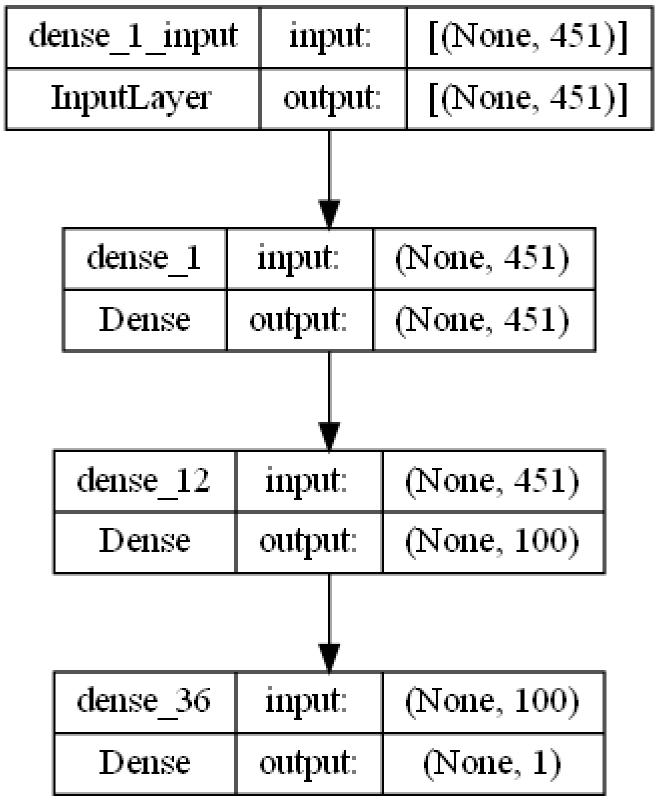

dense_1_input | input: | [(None, 451)]
InputLayer | output: | [(None, 451)]
dense_1 | input: | (None, 451)
Dense | output: | (None, 451)
dense_12 | input: | (None, 451)
Dense | output: | (None, 100)
dense_36 | input: | (None, 100)
Dense | output: | (None, 1)

Figure 2. A graph showing model B's shape.

models; the specific number is arbitrary. The figure 2 depicts the plot, showing the model's structure.

- Model variant C – model, containing 7 hidden layers, each containing 100 neurons. The figure 3 depicts the plot, showing the model's structure.

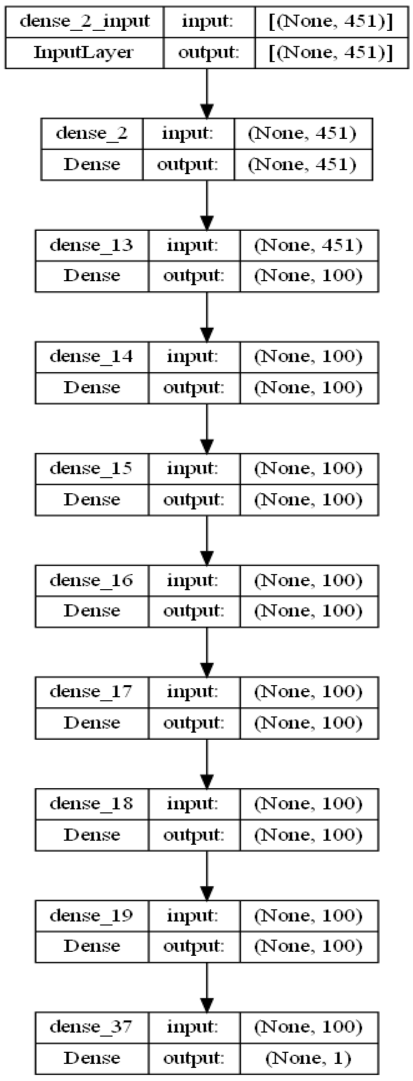

dense_2_input | InputLayer | input: [(None, 451)] | output: [(None, 451)]
dense_2 | Dense | input: (None, 451) | output: (None, 451)
dense_13 | Dense | input: (None, 451) | output: (None, 100)
dense_14 | Dense | input: (None, 100) | output: (None, 100)
dense_15 | Dense | input: (None, 100) | output: (None, 100)
dense_16 | Dense | input: (None, 100) | output: (None, 100)
dense_17 | Dense | input: (None, 100) | output: (None, 100)
dense_18 | Dense | input: (None, 100) | output: (None, 100)
dense_19 | Dense | input: (None, 100) | output: (None, 100)
dense_37 | Dense | input: (None, 100) | output: (None, 1)

Figure 3. A graph showing model C's shape.

- Model variant D – consists of 14 hidden layers with 100 neurons each.

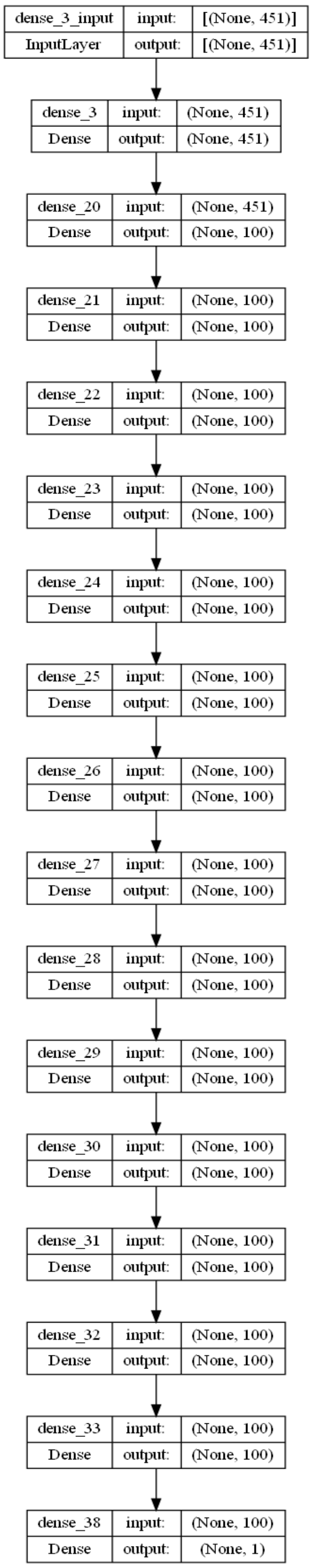

dense_3_input | InputLayer | input: [(None, 451)] | output: [(None, 451)]
dense_3 | Dense | input: (None, 451) | output: (None, 451)
dense_20 | Dense | input: (None, 451) | output: (None, 100)
dense_21 | Dense | input: (None, 100) | output: (None, 100)
dense_22 | Dense | input: (None, 100) | output: (None, 100)
dense_23 | Dense | input: (None, 100) | output: (None, 100)
dense_24 | Dense | input: (None, 100) | output: (None, 100)
dense_25 | Dense | input: (None, 100) | output: (None, 100)
dense_26 | Dense | input: (None, 100) | output: (None, 100)
dense_27 | Dense | input: (None, 100) | output: (None, 100)
dense_28 | Dense | input: (None, 100) | output: (None, 100)
dense_29 | Dense | input: (None, 100) | output: (None, 100)
dense_30 | Dense | input: (None, 100) | output: (None, 100)
dense_31 | Dense | input: (None, 100) | output: (None, 100)
dense_32 | Dense | input: (None, 100) | output: (None, 100)
dense_33 | Dense | input: (None, 100) | output: (None, 100)
dense_38 | Dense | input: (None, 100) | output: (None, 1)

Figure 4. A graph showing model D's shape.

- Model variant E – a model with only 1 hidden layer, but with 902 neurons, equal to the double amount of input features or neurons within the input layer.

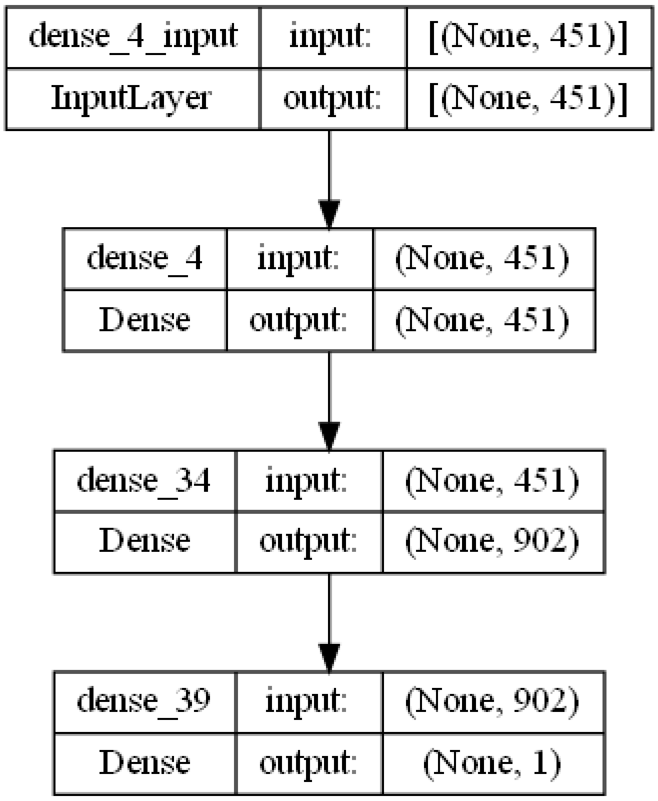


Figure 5. A graph showing model E's shape.

If model B is taken as a baseline, model C provides an ability to determine if only increase of hidden layers affects the performance of the model, considering the equal count of neurons in layers. Model C also has an equal number of layers to the Model A, providing and outlook into the influence of a more intricate shape on the model’s accuracy score. Model D double the number of layers over model C, possibly showing if the increased count of hidden layers has an effect on model’s performance. Model E has only 1 hidden layer, similarly to the model B, but it has a significant increase in the count of neurons, allowing to observe influence from such a change.

Total count of models produced by the solution is 10. The first set of models of variants from A to E are trained based of the “initial” ransomware dataset with 250 samples in combination with benign dataset. The second set of models is trained on the extended malicious dataset with 500 samples in combination with the benign dataset. The models were trained on the main machine with Windows OS without GPU (Graphical Processing Unit) acceleration.

3.5 Evaluation phase

Evaluation and validation of the created model happens during the evaluation phase [9], [10]. The evaluation of created model (proof-of-concept) solution will be covered in the 5th chapter of the thesis “Analysis and validation of the practical implementation”, based on the produced metrics.

Two separate parts of the solution are evaluated:

- Feature extraction through the time taken to collect and convert features from the PE targets.
- Performance of the defined AI models.

The metrics utilised for AI model evaluation are the following:

- Training accuracy – model’s accuracy against the training dataset at a specific epoch.
- Training loss – model’s error rate against the training dataset at a specific epoch.
- Validation accuracy – model’s accuracy against the validation subset at a specific epoch during the training process.
- Validation loss - model’s error rate against the validation subset at a specific epoch during the training process.
- Accuracy score against the testing subset – accuracy of the model, based on the comparison between the produced predicted values by the trained model and the predefined testing subset.
- Confusion matrix – the matrix shows model’s true negative, false negative, true positive and false positive rate.

- ROC (Receiver Operating Characteristic) curve – plots detection system’s false positive rate against their associated true positive rate at various thresholds [3].

- AUC (Area Under the Curve) value – the value depicting model’s accuracy based on the ROC curve’s form, it represents all the threshold values of the curve, by taking the area under it [3].

3.6 Deployment phase

Within the deployment phase, conclusion of the thesis is to be determined within the 6th chapter “Conclusions”, if the techniques utilised and final solution show promise in solving the set-out problem .

4 Practical implementation

Practical solution’s Python scripts, ransomware archives, dataset files, produced plots and serialised data are stored and available within Git repository, link to which is provided within the “Appendix 2 – Solution repository”. The appendix and repository’s “README” file contain a detailed description of repository’s structure.

4.1 Tools and libraries utilised

The cross-platform library “Library to Instrument Executable Formats” (LIEF) is utilised to parse the target ransomware and benign software samples [15]. Keras framework over Tensorflow library (version 2.10.0) is used to define the neural network model. The proof-of-concept is developed in Python version 3.9. Python was chosen for its simplicity and thus suitability of proof-of-concept development and presence of scientific and machine learning libraries, providing tool for AI development. Tensorflow was chosen due to familiarity and experience with the library. The full list of libraries is defined within the “Appendix 3 - Utilised libraries”.

4.2 Data pre-processing and classification

A custom feature extraction algorithm was implemented, which encodes a set of metadata from the target PE binary file into a set of scalar values, which are saved into a CSV dataset. The produced datasets are used in the training process of the neural network model.

The features extracted represent either metadata stored within PE headers themselves, or calculates and encodes properties, such as average of section sizes or a ratio of sections, containing executable code. There were not any strong assumptions on the influence of any particular property on to the efficiency of the produced AI models, thus the easily accessible and simple to calculate properties were chosen to align with one of the thesis’s goals of producing a fast solution.

The number of features was limited to 451 with a consideration for the limited size of the dataset of total 750 or 1000 samples, to limit model's possible performance degradation due to a possible lack of training samples. The feature extraction is defined within Python "feature_extractor.py" script, residing within the repository linked within the Appendix 2.

The following 12 Boolean properties of targets are extracted and encoded in values of 0 (represents "false" value) or 1 (represents "true"):

- has_configuration – true if the PE has a Load Configuration.
- has_debug - true if the PE has a Debug section.
- has_exceptions - true if the PE is using exceptions.
- has_exports - true if the PE has any exported symbol.
- has_imports - true if the PE has imports.
- has_nx - true if the PE has the No-Execute protection.
- has_relocations - true if the PE has relocation entries.
- has_resources - true if the PE has any resources.
- has_rich_header - true if a rich header is present at the end of "dos_stub" and it contains information about the build environment [14].
- has_signature - true if the PE is containing the Authenticode digital signature.
- has_tls - true if the PE is using thread local storage.

Two sections of code dealing with Boolean value extraction are shown in the following figures 6 and 7.

```
# A list of PE's properties, that are expressed as a boolean value within the
Binary class from LIEF.
pe_properties = ['has_configuration',
               'has_debug',
               'has_delay_imports',
               'has_exceptions',
               'has_exports',
               'has_imports',
               'has_nx',
               'has_relocations',
               'has_resources',
               'has_rich_header',
               'has_signatures',
               'has_tls',
]
```

Figure 6. A code section defining a list of Boolean properties under extraction.

```
# Extract and encode target's PE properties. 12 features/point vector will be
produced.
print("Extracting and encoding target's properties.")
pe_properties_vector = np.array([0.0] * len(pe_properties))
for index, pe_property in enumerate(pe_properties):
    pe_properties_vector[index] = 1.0 if getattr(pe_binary, pe_property) else
0.0
```

Figure 7. A code section defining Boolean property extraction and encoding.

The first 64 bytes of the PE entry point function were extracted as they can help the model to identify binaries with more distinctive entry-points. Values of bytes vary between 0 and 255, the extracted values are normalized to values within [0.0,1.0] range, by dividing each of them by 255. The related section of code dealing with entry-point extraction is shown in the following figure 8.

```
# Encode target's entrypoint. Normalizing to [0.0,1.0]. 64 features/point
vector will be produced.
print("Encoding target's entrypoint.")
# Pad the array if the amount of extracted bytes is less than 64.
while len(entry_bytes) < 64:
    entry_bytes += [0.0]
# Store the array of bytes in form of a normalized array.
entry_bytes_vector = np.array(entry_bytes) / 255.0
```

Figure 8. A section of code defining entry-point extraction and encoding.

A histogram of the of each byte repetitions is produced, resulting in a list of 256 values. The histogram will encode a form of binary's statistical representation by noting the frequency of certain bytes' value appearance within the binary file. The related section of code dealing with histogram creation is shown in the following figure 9.

```
# Extract and encode target's histogram of the normalized histogram of each
byte frequency.
# 256 features/point vector will be produced.
# Count the number of occurrences of each value in the target's file/array.
print("Extracting and encoding target's histogram")
histogram = np.bincount(np.frombuffer(raw_file, dtype=np.uint8),
minlength=256)
histogram = histogram / histogram.sum()
```

Figure 9. A section of code defining histogram creation over target's raw bytes.

A set list of imported functions is defined, based on the list of important Windows functions, which are often encounter by malware analysts, defined within the chapter “A: Important Windows Functions” from the book “Practical Malware Analysis” [6]. The PE binaries are checked for the import of 114 functions, which are often utilised by malicious software, thus their presence may indicate a possible malicious intent. The related section of code dealing with import function creation is shown in the following figure 10.

```
# Extract and encode imported functions. 114 features/point vector will be
produced.
print("Extracting and encoding imported functions.")
targets_extracted_imported_functions = [imported_function.name for
imported_function in pe_binary.imported_functions]
targets_imported_functions_vector = np.array([0.0] * len(imported_functions))
for index, imported_function in enumerate(imported_functions):
    if imported_function in targets_extracted_imported_functions:
        targets_imported_functions_vector[index] = 1.0
```

Figure 10. A section of code dealing with encoding of imported functions.

The size of the PE binary on disk and in memory or its raw size against its defined virtual size. Raw size of a PE section is the true size taken by the section within the binary file. Virtual size represents the number of bytes that will be mapped to the loaded section. These values may differ, as the bigger virtual size over the raw size indicates presence of uninitialized values. The produced amount of produced features is 1. The related section of code dealing with import function creation is shown in the following figure 11.

```
# Encode size. 1 features/point vector will be produced.
print("Encoding size.")
size_vector = [min(file_size, pe_binary.virtual_size) / max(file_size,
pe_binary.virtual_size)]
```

Figure 11. A section of code dealing with size encoding.

A set of 4 features was calculated based on the PE’s properties, the related code is depicted with the figure 12:

- An average ratio of the PE sections containing code in relation to the total amount of sections.
- The ratio of sections marked as executable against the rest.
- The average entropy among the PE sections.
- Average ratio of section's raw size within file against the defined virtual size.

```
# Extract and encode sections. 4 features/point vector will be produced.
print("Extracting and encoding sections.")
# Create a list of dictionaries, corresponding to existing PE sections within
the binary
# and describing their characteristics.
pe_sections = [
    {
    'characteristics': ','.join(map(str, section.characteristics_lists)),
    'entropy': section.entropy,
    'name': section.name,
    'size': section.size,
    'vsize': section.virtual_size
    } for section in pe_binary.sections
]

num_sections = len(pe_sections)
max_entropy = max([section['entropy'] for section in pe_sections]) if
num_sections else 0.0
max_size = max([section['size'] for section in pe_sections]) if num_sections
else 0.0
min_vsize = min([section['vsize'] for section in pe_sections]) if
num_sections else 0.0
norm_size = (max_size / min_vsize) if min_vsize > 0 else 0.0

sections_vector = [
    # Calculate an average ratio of the PE sections containing code in the
relation to the total amount of sections.
    (len([section for section in pe_sections if
'SECTION_CHARACTERISTICS.CNT_CODE' in
          section['characteristics']]) / num_sections) if num_sections else
0,
    # Calculate the ratio of sections marked as executable vs the rest.
    (len([section for section in pe_sections if
'SECTION_CHARACTERISTICS.MEM_EXECUTE' in
          section['characteristics']]) / num_sections) if num_sections else
0,
    # Calculate the average entropy among the PE sections.
    ((sum([section['entropy'] for section in pe_sections]) / num_sections) /
     max_entropy) if max_entropy > 0 else 0.0,
    # Calculate average ratio of section's size vs their virtual size.
    ((sum([section['size'] / section['vsize'] for section in pe_sections]) /
num_sections) /
     norm_size) if norm_size > 0 else 0.0,
]
```

Figure 12. A section of code depicting encoding of PE's sections metadata.

The compiled dataset contains an additional column of values – the label. The label, is determined manually, depending on the target under feature extraction. If the target is ransomware the value equal 1, 0 otherwise. The label column is separated from the dataset during the training process, as it will be used for validation of the trained model's accuracy.

4.3 Data pre-processing time performance

Data extraction and encoding are a significant part of the solution and significantly contribute to the total performance of the solution. The wall time (time taken from start to finish, even if the computer did not process the target task all the time) and the CPU time (time that was spent only on the processing of the task) were measured to determined minimum, maximum, average, median and total times (in milliseconds) taken to extract features. The measured values will be reference within the 5th chapter "5 Analysis and validation of the practical implementation".

The following table contains the measured time values during feature extraction from the initial ransomware dataset (250 samples):

	Wall time (milliseconds)	CPU time (milliseconds)
Average time taken	37.44	37.68
Minimum time taken	1.56	1.56
Maximum time taken	370.64	370.43
Median time taken	23.82	23.75
Total time taken	9363.09	9422.68

Table 1. Table containing measured time values for the feature extraction from initial ransomware dataset.

The following table contains the measured time values during feature extraction from the extended ransomware dataset (500 samples):

	Wall time (milliseconds)	CPU time (milliseconds)
Average time taken	71.71	71.82
Minimum time taken	1.49	1.49
Maximum time taken	1551.59	1550.95
Median time taken	27.25	27.22
Total time taken	35856.79	35915.48

Table 2. Table containing measured time values for the feature extraction from extended ransomware dataset.

The following table contains the measured time values during feature extraction from the extended benign software dataset (500 samples):

	Wall time (milliseconds)	CPU time (milliseconds)
Average time taken	6586.19	6434.03
Minimum time taken	10.71	15.63
Maximum time taken	416483.55	416468.75
Median time taken	851.28	671.87
Total time taken	3293100.87	3217046.88

Table 3. Table containing measured time values for the feature extraction from benign software dataset.

4.4 Model definition specifics and training

Model input and hidden layers utilise ReLU (Rectifier linear unit activation function) activation function. Purpose of an activation function is to apply a non-linear transformation to the neural network's parameter [3]. ReLU allows for fast learning and is more resilient against vanishing gradient problem, which means the parameter's contribution or significance is more resilient throughout training process [3], [11].

The following section of code defines the initial input layer for all the 5 models in form of figure 13:

```
# Define and add an input layer.
for ai_model in models.values():
    ai_model.add(keras.layers.Dense(units=input_amount, activation='relu',
input_shape=X_train[0].shape))
```

Figure 13. A section of code depicting addition of an input layer to the AI models.

The final, output layer of the models utilises a sigmoid activation function, which produced values only within the range of [0, 1]. Such form of output is suitable for binary classification (or classification of the target) between two classes (or labels). Though, it is less resilient against vanishing gradient problem, this drawback is overcome by the application of the function to the output layer [3], [11]. The output layer's definition is shown with the figure 14.

```
# Define and add an output layer.
for ai_model in models.values():
    ai_model.add(keras.layers.Dense(units=1, activation='sigmoid'))
```

Figure 14. A section of code defining AI model's output layer.

During model's compilation, additional model's parameters are set, configuring the training process. The models are configured to utilise Adam optimizer, binary cross-entropy loss function and to output accuracy metrics of the model under the training process throughout the epochs (or training iterations).

Optimizers are algorithms, used to determine how the loss or differences between predicted and actual values, produced during the training process will affect the modification of AI model's parameters. Adam algorithm is often chosen as the starting algorithm due to its general good performance [11].

The loss function is a function, used to represent the difference between the true training labels and model's predicted labels. The value produced by the function indicates the model's quality, during the training process the loss value is attempted to be minimized. The binary cross-entropy loss function is suitable for model attempt to identify the target between two classes (or labels) [3], [11]. The models' training configuration is shown with the figure 15.

```
# Compile the model and train it.
# Define an Adam optimizer, binary crossentropy for the loss function and
metrics to be shown.
for ai_model in models.values():
    ai_model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])
```

Figure 15. A section of code, compiling the AI model.

The model is trained upon the 70% of the dataset set out for training process and validated during the training process with a separate 15% of the dataset entries. The default amount of epoch to undergo is set to an arbitrarily high number of 10000. To prevent possible overfitting, a callback is defined, which monitors model's performance after each training iteration, observing model's validation loss (or error rate against the validation subset) with a patience of 100 epochs. The callback rolls back the model's parameters, which performed the best over the last 100 epochs, thus, potentially, stopping the training process before going through all of the 10000 iterations with an aim to preserve the best version of the model. The models' training definitions are shown with the figures 16 and 17.

```
# A callback that will monitor the state of the model and stop the training
process prematurely to avoid over-fitting.
# In "min" mode the end for the training will be considered, when the tracked
value stops decreasing.
earlystopper = keras.callbacks.EarlyStopping(monitor="val_loss", mode="min",
patience=100, restore_best_weights=True, verbose=1)
```

Figure 16. A section of code, showing the definition of a callback function for the training process.

```
history = ai_model.fit(x=X_train, y=y_train, validation_data=(X_validate,
y_validate), epochs=amount_of_epochs, callbacks=[earlystopper])
```

Figure 17. A section of code, showing AI model's training process definition.

4.5 Validation results of the trained models

The following metrics were calculated for all the 10 models (all 5 variants, against the initial and extended malicious datasets): training accuracy, training loss, validation accuracy, validation loss, accuracy score against the testing subset, confusion matrix, epochs taken to produce the most performant model, ROC curve and an AUC value, with the wall and CPU times taken for their training. Within the current chapter accuracy scores and AUC value are presented within the Table 4, the rest of the metrics are depicted in for of the plots and can be found in the “Appendix 4 – Trained models’ metrics and plots”. All the values within the Table 4 are rounded to two decimal places.

	Accuracy score	AUC value	Wall time (milliseconds)	CPU time (milliseconds)	Epochs taken
Model A (initial ransomware dataset)	0.92	0.93	8750.39	100812.5	6
Model A (extended ransomware dataset)	0.87	0.87	14401.00	168515.63	65
Model B (initial ransomware dataset)	0.88	0.89	6510.73	76671.88	4
Model B (extended ransomware dataset)	0.87	0.87	7005.46	84562.5	4
Model C (initial ransomware dataset)	0.89	0.88	8965.27	105000.00	4
Model C (extended ransomware dataset)	0.92	0.92	9668.04	116359.38	5

	Accuracy score	AUC value	Wall time (milliseconds)	CPU time (milliseconds)	Epochs taken
Model D (initial ransomware dataset)	0.88	0.91	19929.75	241015.63	89
Model D (extended ransomware dataset)	0.87	0.87	12712.49	152078.13	6
Model E (initial ransomware dataset)	0.91	0.92	8744.34	103750.00	4
Model E (extended ransomware dataset)	0.88	0.88	8783.94	109796.88	4

Table 4. A table, containing model's accuracy score and AUC values.

5 Analysis of AI model's validation results

From the produced accuracy metrics and plots for the defined AI models the following key point can be noticed:

- A simple feed-forward neural network model shows promising results in detection of ransomware PE binaries against the benevolent ransomware. As the the model's accuracy score varies from 0.88 to 0.92 (1 being the best possible score, and 0 – the worst) for models trained on 250 samples of ransomware (750 samples total), and 0.88 to 0.92 on the extended dataset (1000 samples total), which are promising, considering the limited number of static features collected and the limited size of the dataset.

- Differing shapes of the models, though, behaved differently throughout the training process (accuracy and loss plots can be found in the "Appendix 4 – Trained models' metrics and plots"), produced similarly accurate predictions from their best restored version, from the training process. The similarity in results also reinforces promising applicability of the current approach for ransomware detection – usage of a neural network for ransomware detection, based on the target's simple static features.

- The extension of the dataset did result in subtle degradation of A, B, D and E models accuracy scores by 0.01 or 0.03 points. Model C saw a small increase in accuracy by 0.03 points. As it can be seen, the difference in models' performance between the datasets is small and its form of influence varies between model shapes, thus it cannot be definitively considered, if the dataset's extension is beneficial or detrimental in current case. For a more precise answer, the dataset should be significantly increased to cover wider range of software, which could result in more drastic changes.

- The average time taken in milliseconds for feature extraction from malware is 37.68 and 71.82 for initial, and extended datasets respectively. On average, the extraction of PE's features for benign samples has taken 6434.03 milliseconds or

6.43 seconds. These results show that the current feature extraction method can quickly collect the required data for model training or for the target to be analysed by the trained model.

- AI model average training wall (or real) time varied from 6510.73 to 19929.75 milliseconds or 6.51 or 19.93 seconds. These measurements shows the swiftness of the current model's training process. The training process was not GPU accelerated.

6 Conclusions

In the thesis, the applicability of a simple feed-forward neural network, trained on the target PE binaries' static features was investigated. The retrieved accuracy of the defined AI models varies from 0.88 to 0.92 accuracy score. Considering the limited number of features defined (451 features) and the size of the dataset (750 or 1000 total samples), this fact shows that the neural networks are applicable for ransomware detection, based on the simple extracted static features.

In addition, the resulting solution is fast to prepare (in terms of feature generation) and train, providing an advantage over more intricate and complex alternative implementations [4]. This could prove beneficial for the fast detection and prevention of target binary's execution, as the execution of the ransomware should be prevented as soon as possible to prevent the cause of damage to the affected system, which can be devastating by the nature of ransomware.

This research shows that the neural networks are a promising option in ransomware binary detection, even with the limited feature set and datasets.

7 Future work

The future work could take the current topic into the following directions:

- Development of a more precise and production-ready solution. Higher accuracy could be potentially achieved with bigger dataset and further extraction enhancements. The solution could be made to automatically monitor the target system for appearance of new binaries, which would be immediately scanned by the solution.

- An alternative algorithm can be investigated with the goal of ransomware detection based on the target's static features, such as CNN (Convolutional Neural Network). The CNN has proven itself at image recognition [3]. To utilise the algorithm, binaries could be converted into images, representing their static properties, which could then be analysed by the CNN model.

- An anti-malware suit could be developed, consisting of multiple AI models, monitoring separate aspects of the system, such as, static features of the files, dynamic features of programs' executions, network traffic, with an aim of all-encompassing monitoring against possible malware infection.

- A GAN (Generative Adversarial Network) system could be tested for ransomware detection. The system consists of two neural networks, competing against each other with an aim to improve their accuracy. The generative network's task is to generate fake samples. Discriminator network's goal is to distinguish the difference between the real values and generated ones [3]. The model could be tested for generation artificial samples, representing ransomware, with an aim to make the network more resilient against modified variations of the same ransomware and, possibly, yet unseen variants.

References

[1] Statista Research Department, “Annual number of ransomware attacks worldwide from 2016 to first half 2022”, August 2022 https://www.statista.com/statistics/494947/ransomware-attacks-per-year-worldwide/ (accessed January 5, 2023)

[2] Internet Crime Complaint Center IC3, “2021 IC3 Annual Report” https://www.ic3.gov/Media/PDF/AnnualReport/2021_IC3Report.pdf (accessed January 5, 2023)

[3] Joshua Saxe, Hillary Sanders “Malware Data Science” No Starch Press, September 2018

[4] Damien Warren Fernando, Nikos Komninos, Thomas Chen “A Study on the Evolution of Ransomware Detection Using Machine Learning and Deep Learning Techniques”, September 2020

[5] Microsoft “Defining Malware: FAQ”, April 2009 https://learn.microsoft.com/en-us/previous-versions/tn-archive/dd632948(v=technet.10)?redirectedfrom=MSDN (accessed January 5, 2023)

[6] Michael Sikorski, Andrew Honig “Practical Malware Analysis” No Starch Press, February 2012

[7] Allan Liska, Timothy Gallo “Ransomware: Defending Against Digital Extortion” O’Reilly Media: Sebastopol, 2017

[8] Sophos Knowledge Base: Ransomware: How an Attack Works, 2016 https://community.sophos.com/kb/en-us/124699 (accessed January 5, 2023)

[9] Wikipedia “Cross-industry standard process for data mining”, last time modified August 2022 https://en.wikipedia.org/wiki/ Cross_Industry_Standard_Process_for_Data_Mining (accessed January 5, 2023)

[10] Shearer C. “The CRISP-DM model: the new blueprint for data mining”, J Data Warehousing, 2000

[11] Andrew Glassner “Deep Learning: A Visual Approach” No Starch Press, June 2021

[12] MalwareBazaar “Malware sample exchange” https://bazaar.abuse.ch/ (accessed January 5, 2023)

[13] SourceForge “The Complete Open-Source and Business Software Platform” https://sourceforge.net/ (accessed January 5, 2023)

[14] LIEF Documentation https://lief-project.github.io/doc/stable/index.html (accessed January 5, 2023)

[15] LIEF “Library to Instrument Executable Formats” https://lief-project.github.io/ (accessed January 5, 2023)

[16] J. D. Hunter "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.

[17] Microsoft Documentation “PE format” https://learn.microsoft.com/en-us/windows/win32/debug/pe-format (accessed January 5, 2023)

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Solution repository

Current thesis' solution, its scripts, ransomware archives, plots, models, serialised objects are stored and present on the TalTech's GitLab instance under the link: https://gitlab.cs.ttu.ee/mikara/ransomware-machine-learning-detector. The repository is available for the internal view by the logged in users.

Python scripts are available at the highest level of the repository's directory. The solution consists of two main Python scripts. "feature_extractor.py" one focuses on the feature extraction from Windows Portable Executable binaries and creates a CSV dataset, containing vectors of static features, extracted either from the benign binary targets or malicious ones. "model_trainer.py" script defines the feed-forward, supervised, artificial neural network models and conducts their training, after which the created models are saved in the "model/" directory, which can be used for further calculations, applications, measurements, or experiments. Models are divided between the "extended/" and "initial/" directories, indicating on what malicious dataset the models have been trained. Initial dataset contains 250 samples of ransomware features, while extended has 500 samples. Models are divided by letter in their name to indicate their differences, all the models bear a different internal shape - different count of hidden layers and neurons per layer.

The following scripts do not play part in feature extraction, models definition or their training, they are used for analysis:

- "benign_file_info_extractor.py" - a script creating a dataset of benign files' calculate SHA256 hash, their file name and size in bytes for further analysis.
- "benign_hashes_evaluator.py" - a script calculating an average, mean, minimum and maximum size in bytes of the benign files, based on the created dataset with the "benign_file_info_extractor.py".
- "model_evaluator.py" - a script importing the save models and calculating predictions based on the testing subset, confusion matrix, accuracy score, ROC curve, AUC value. Plots for confusion matrix and ROC curve are also generate

and stored within “plots/” directory either under “extended/” or “initial/” subdirectory, depending on the model.

- “ransomware_hashes_evaluator.py” - a script calculating an average, mean, minimum and maximum of ransomware's detections count by anti-malware vendor; ransomware's size in bytes; and grouping ransomware samples under their defined families/signatures.

Pickles - Python serialised objects. The “model_trainer.py” script stores defined training subsets of features and labels as pickles within the “pickles” directory under the “initial/” or “extended/” subdirectories, depending on the dataset utilised.

The pickles are imported by the “model_evaluator.py” script to conduct prediction on the determined testing subsets and calculate the defined metrics, such as confusion matrix, accuracy score, ROC curve and AUC value.

Datasets are stored under datasets/ directory. malicious_dataset.csv, malicious_dataset_extended.csv and benign_dataset.csv are produced by the feature_extractor.py script and contain extracted features of targets, used for training of the AI models.

“malicious_hashes.csv” and “malicious_hashes_extended.csv” are datasets, stored under “ransomware_hashes/” subdirectory, containing SHA256 hashes, family/signature names, amount of vendor detections, size in bytes and first time seen of the ransomware samples, which were used for creation of the “malicious_dataset.csv” dataset. These sets are used by the “ransomware_hashes_evaluator.py” script for analysis.

“benign_hashes/” directory contains dataset produced by the “benign_file_info_extractor.py” script, containing SHA256 hashes, file names and size in bytes of the benign software samples, based on the category they were taken from, such as: database, internet, scientific, software development, system and a complete dataset, containing all the samples.

The binaries of utilised ransomware samples are also present in the archived form within “ransomware_archived.zip” and “ransomware_archived_extended.zip” archive, under “archive/” subdirectory.

Appendix 3 – Utilised libraries

The solution was developed, tested and run with Python 3.9. “Anaconda” was used for the “conda” virtual environment to set up the required libraries for development. Tensorflow/Keras version 2.10.0 was used for development of the neural network.

Libraries used:

tensorflow/keras - library of neural network definition and training.

os - library for operations on the operating system.

hashlib - library for hash calculations.

csv - library for csv file operations.

pandas - library for data manipulation and analysis.

statistics - library for statistic operations.

lief - library for binary parsing, modification and abstraction.

numpy - library for scientific computing.

time - library for time operations.

sklearn - library for predictive data analysis.

matplotlib - library for plotting, data visualisation [16].

seaborn - library for statistical data visualisation.

pickle - library for Python object (de-)serialisation.

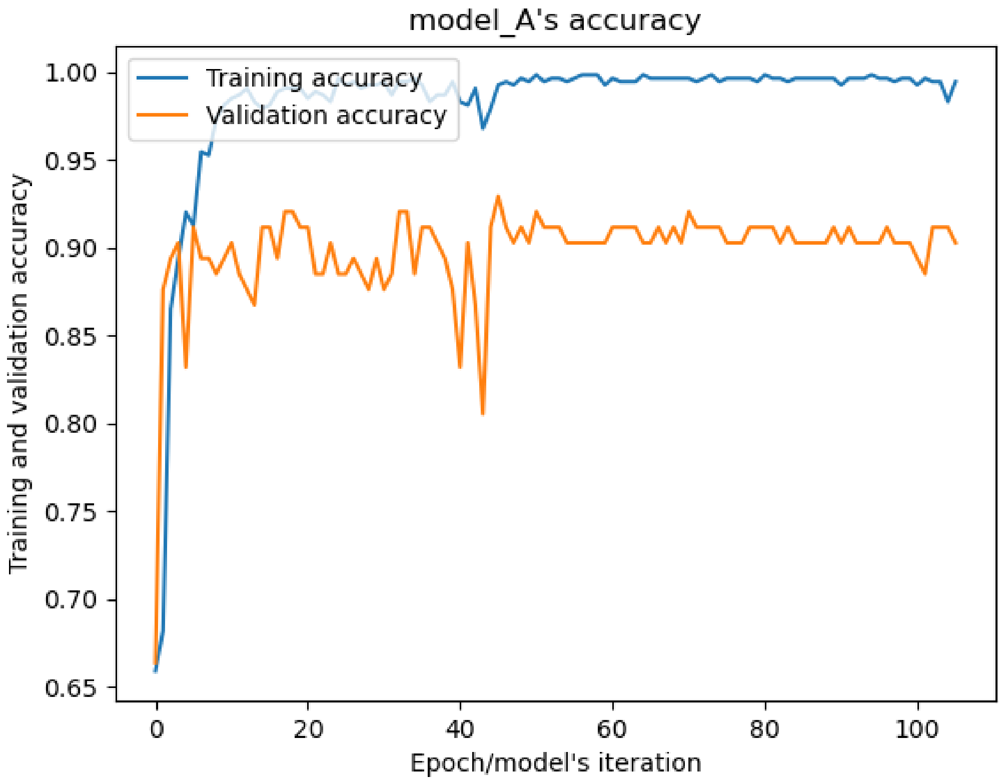
Appendix 4 – Trained models' metrics and plots

Within the current "Appendix", models' plots of training accuracy, training loss, validation accuracy, validation loss, confusion matrix, ROC curve with an AUC value are depicted.

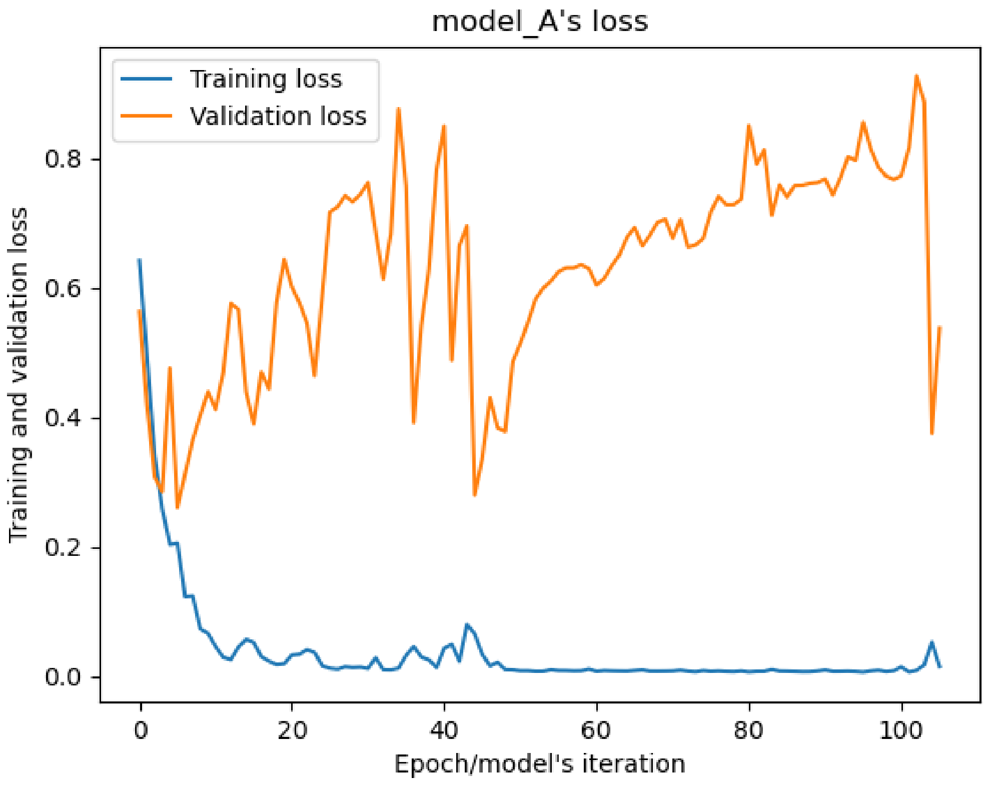
Model A's plots

Trained with the initial ransomware dataset

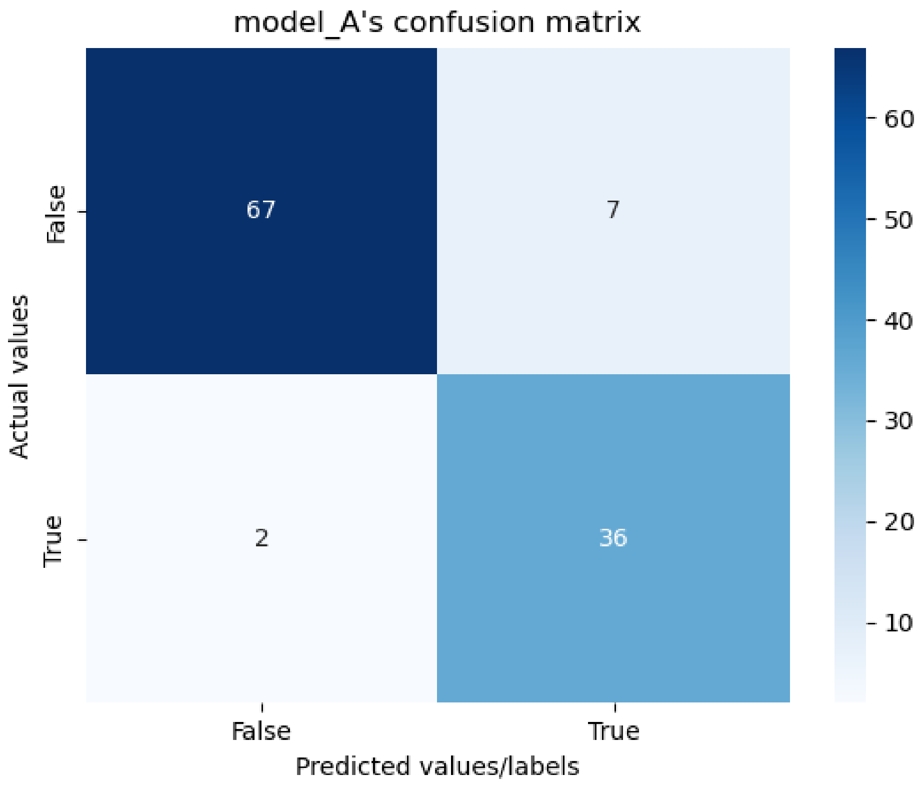
Graph of the model A (trained on initial ransomware dataset)'s accuracy values during the training process:



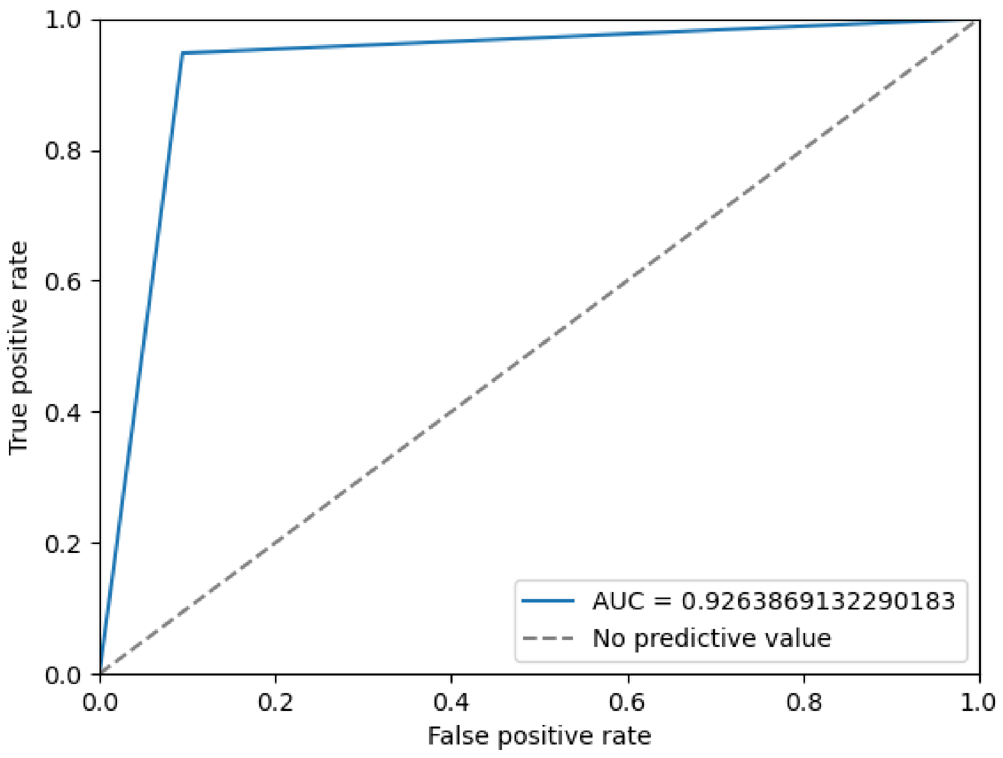
Graph of the model A (trained on initial ransomware dataset)'s loss values during the training process:



Graph of the model A (trained on initial ransomware dataset)'s confusion matrix:

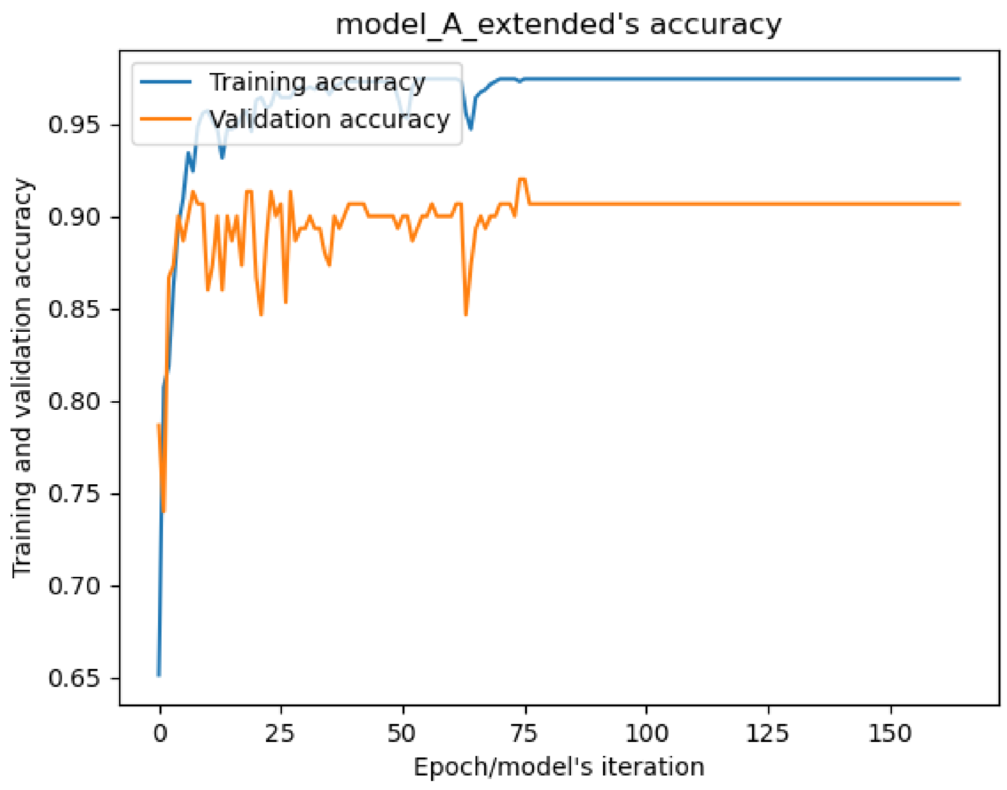


Graph of the model A (trained on initial ransomware dataset)'s ROC curve:

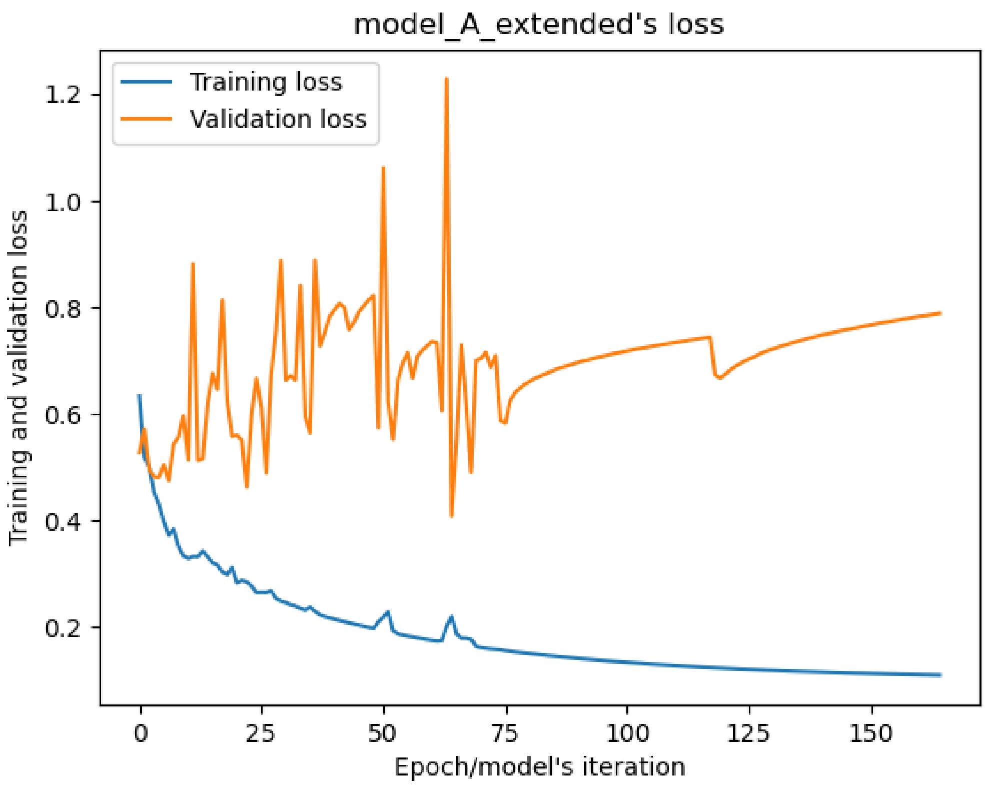


Trained with the extended ransomware dataset

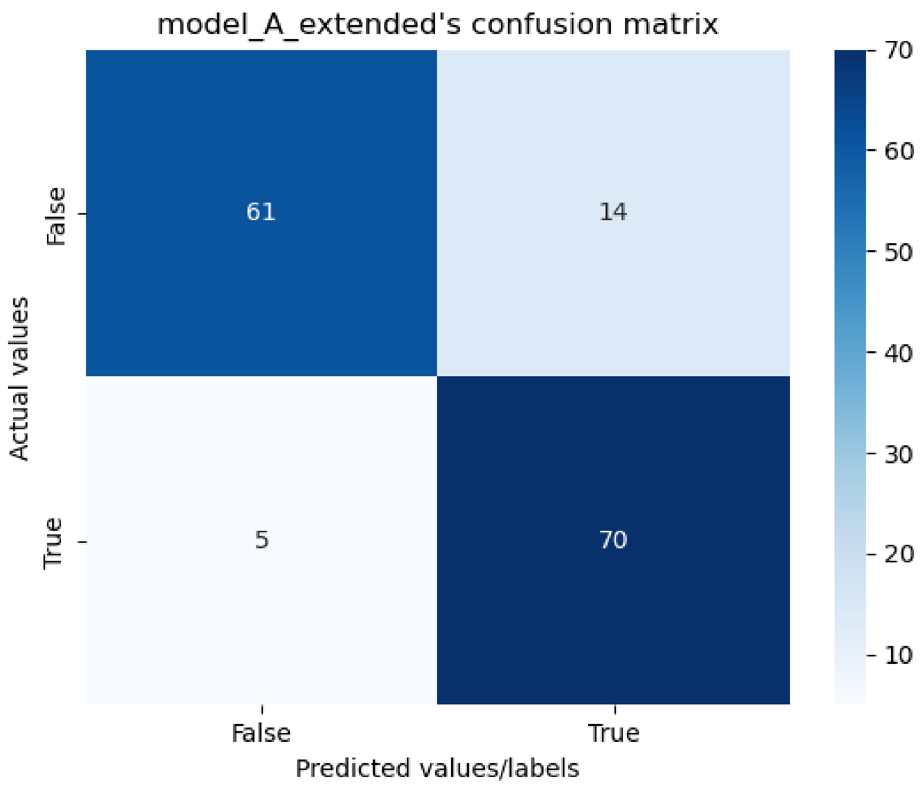
Graph of the model A (trained on extended ransomware dataset)'s accuracy values during the training process:



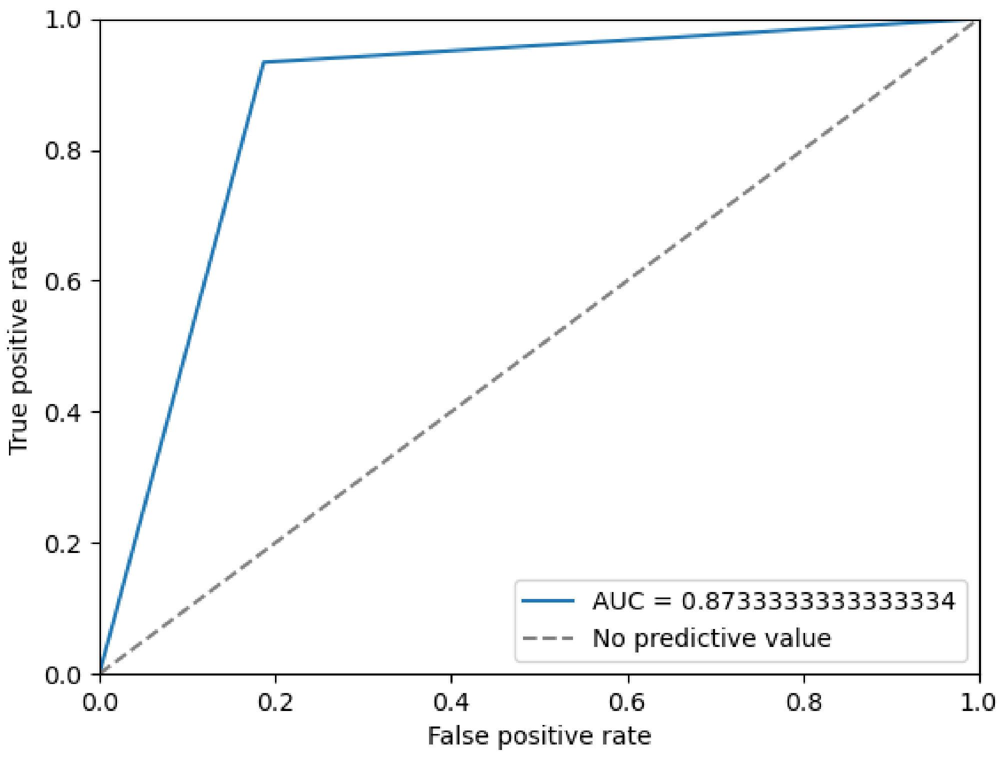
Graph of the model A (trained on extended ransomware dataset)'s loss values during the training process:



Graph of the model A (trained on extended ransomware dataset)’s confusion matrix:



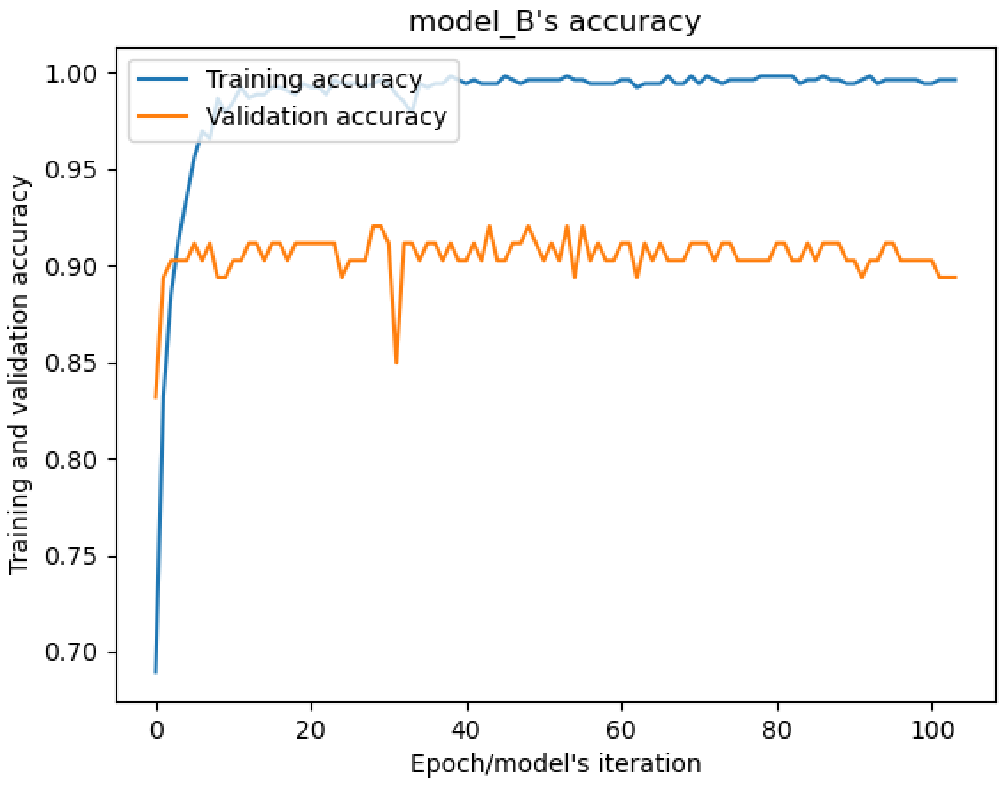
Graph of the model A (trained on extended ransomware dataset)'s ROC curve:



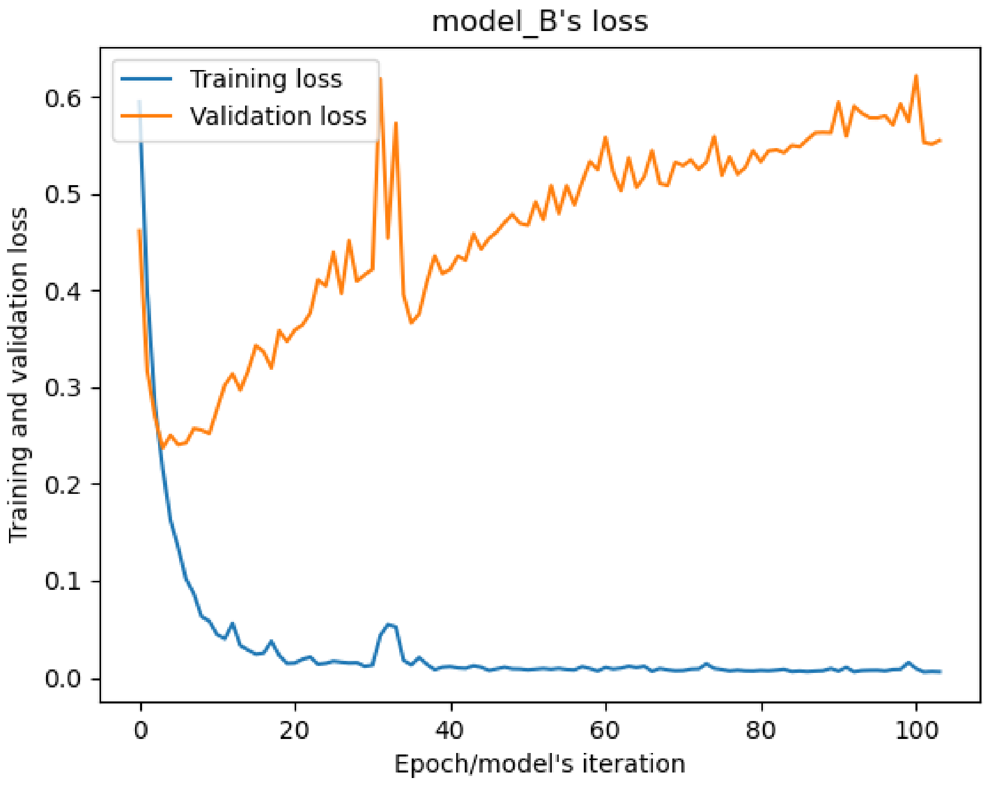
Model B's plots

Trained with the initial ransomware dataset

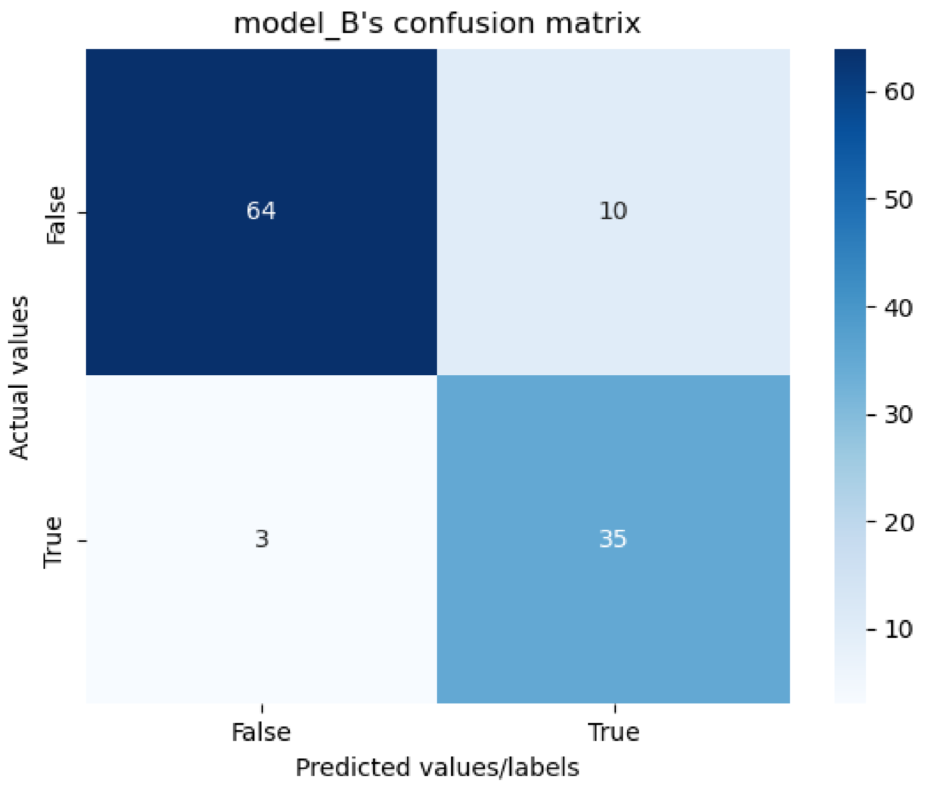
Graph of the model B (trained on initial ransomware dataset)'s accuracy values during the training process:



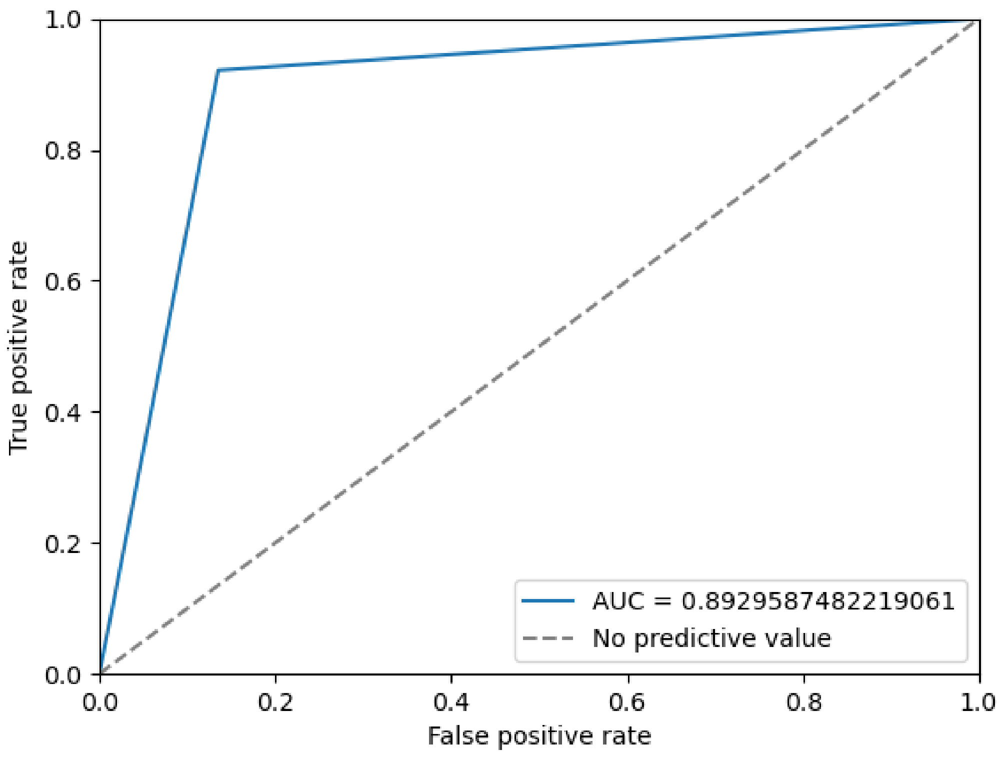
Graph of the model B (trained on initial ransomware dataset)'s loss values during the training process:



Graph of the model B (trained on initial ransomware dataset)'s confusion matrix:

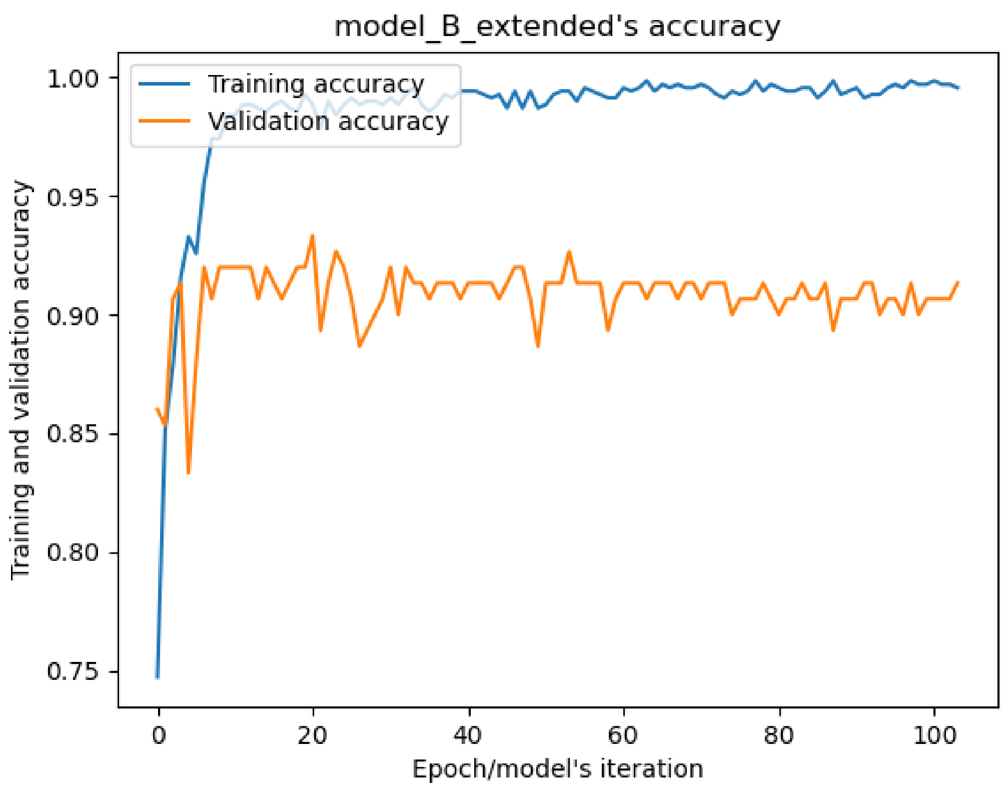


Graph of the model B (trained on initial ransomware dataset)'s ROC curve:

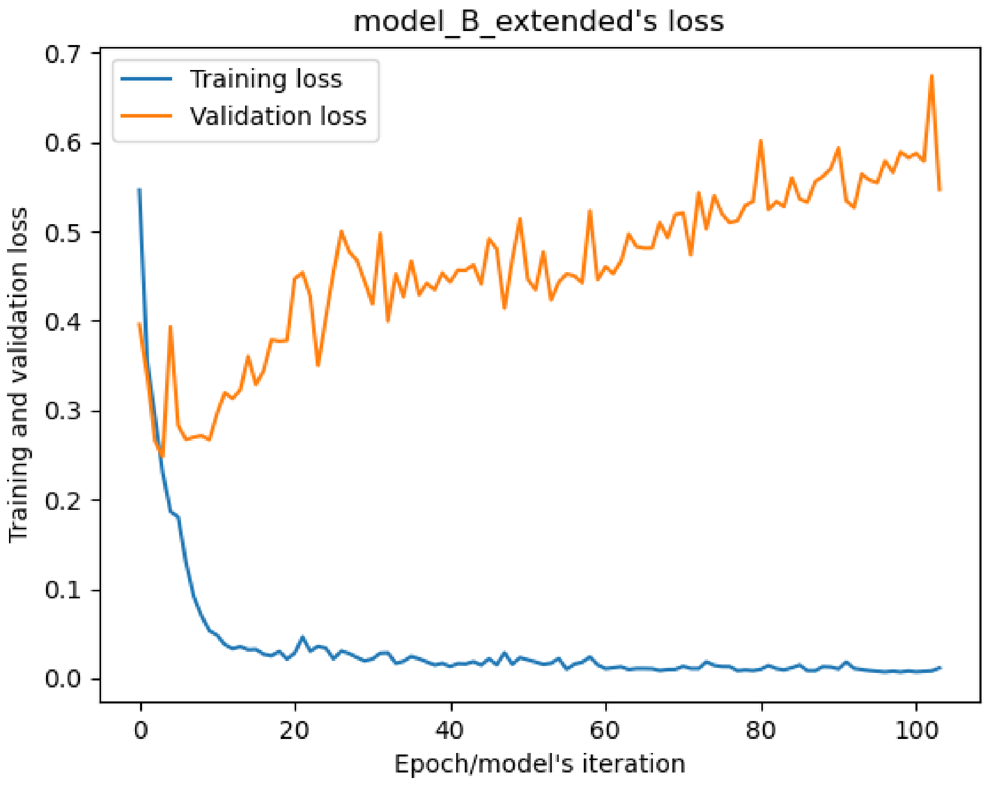


Trained with the extended ransomware dataset

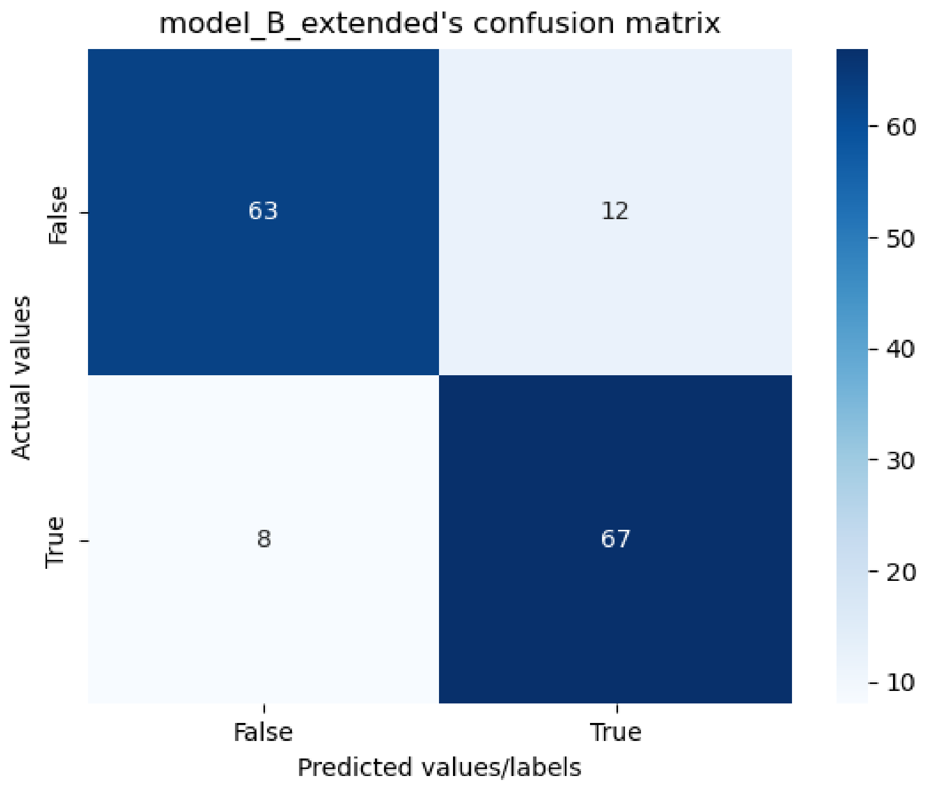
Graph of the model B (trained on extended ransomware dataset)'s accuracy values during the training process:



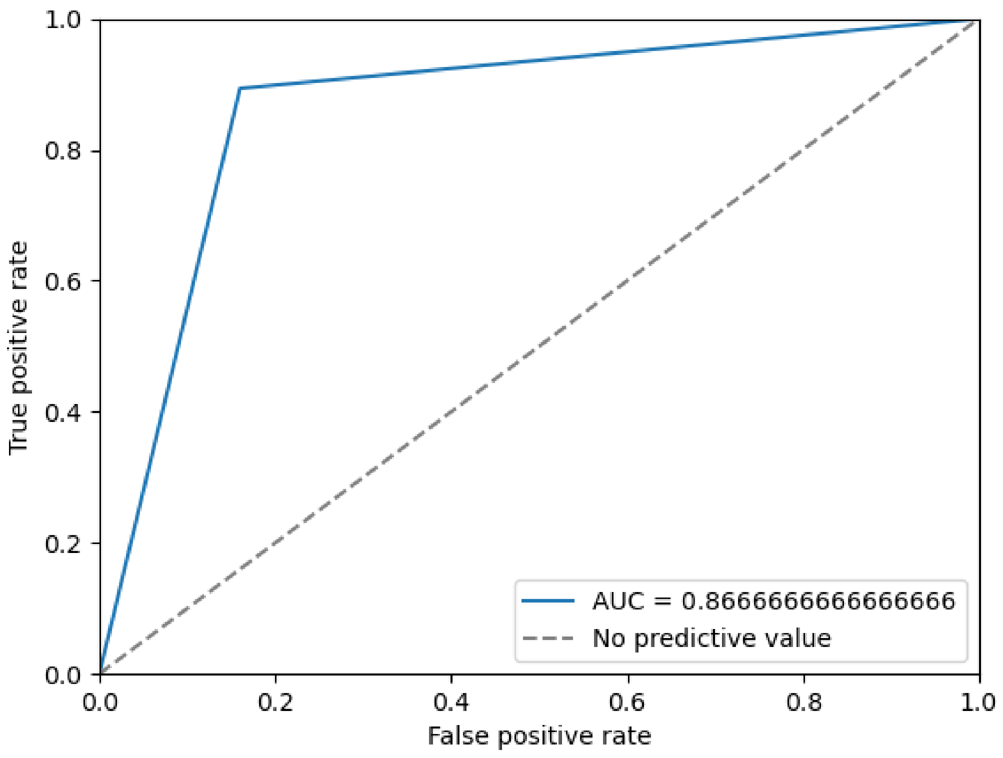
Graph of the model B (trained on extended ransomware dataset)'s loss values during the training process:



Graph of the model B (trained on extended ransomware dataset)'s confusion matrix:



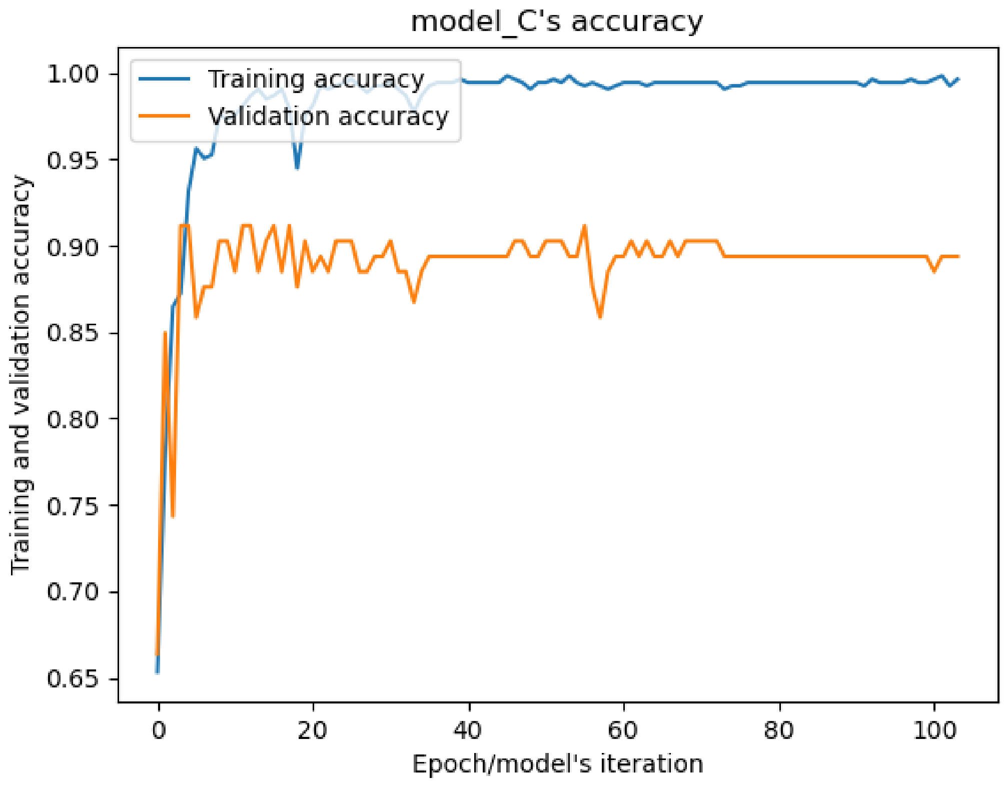
Graph of the model B (trained on extended ransomware dataset)'s ROC curve:



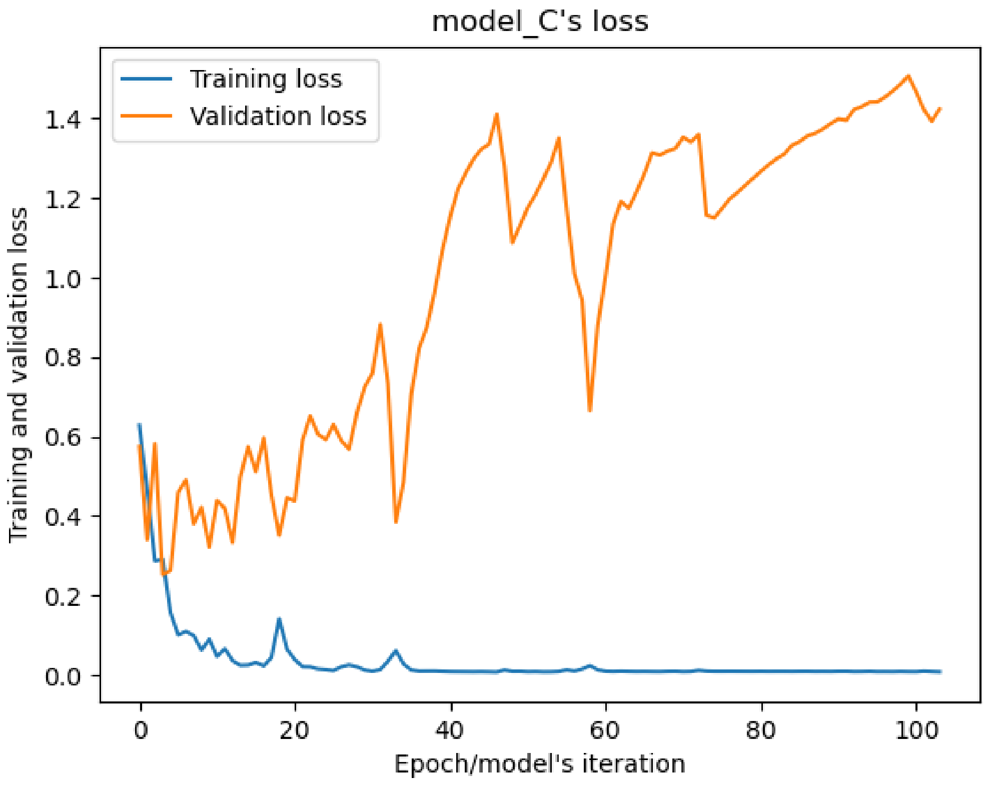
Model C's plots

Trained with the initial ransomware dataset

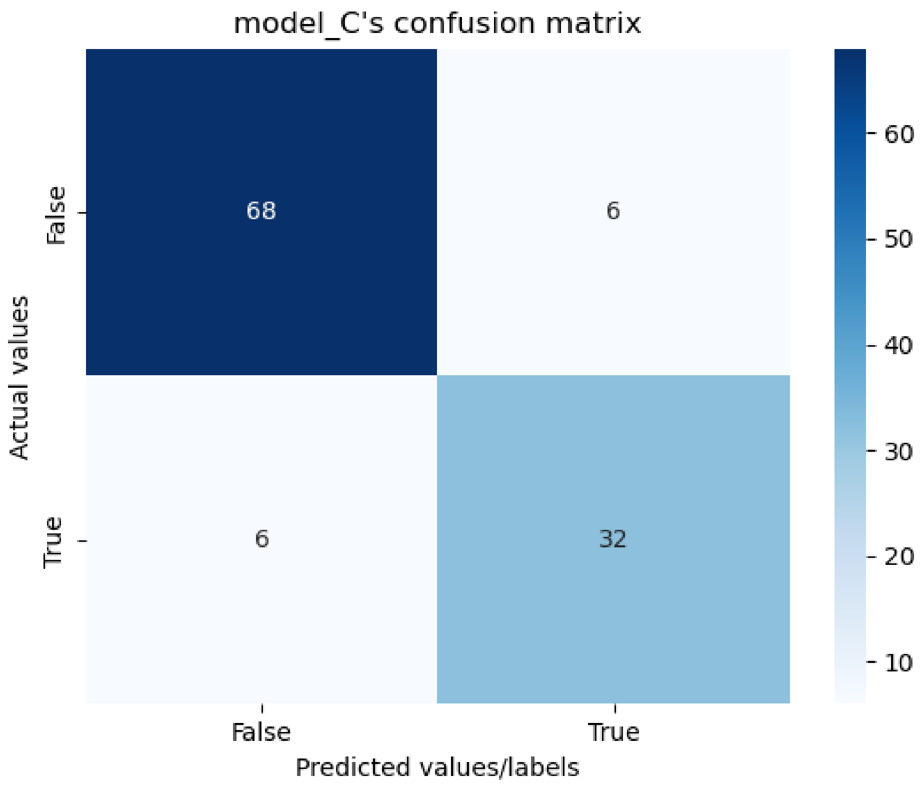
Graph of the model C (trained on initial ransomware dataset)'s accuracy values during the training process:



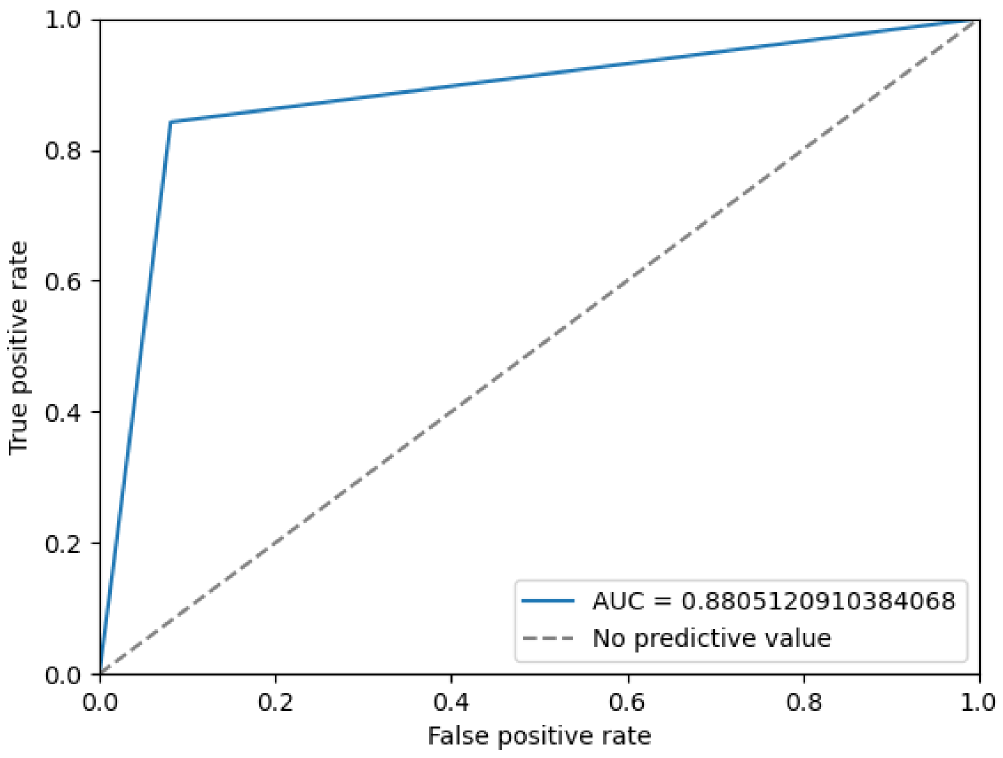
Graph of the model C (trained on initial ransomware dataset)'s loss values during the training process:



Graph of the model C (trained on initial ransomware dataset)’s confusion matrix:

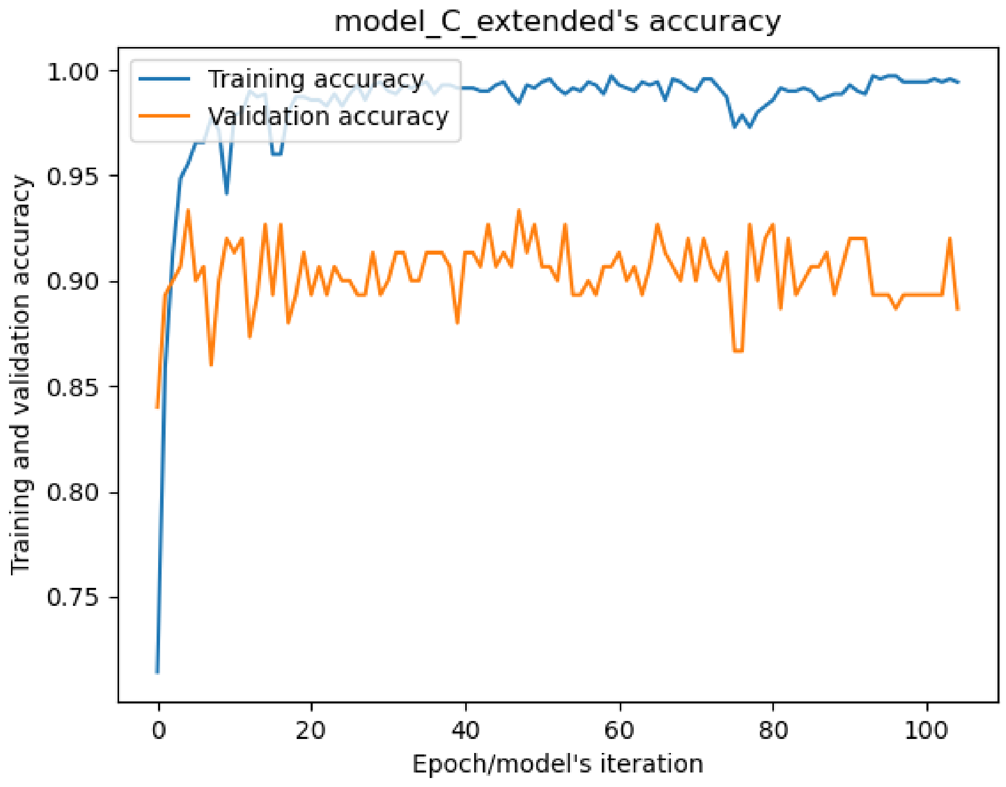


Graph of the model C (trained on initial ransomware dataset)'s ROC curve:

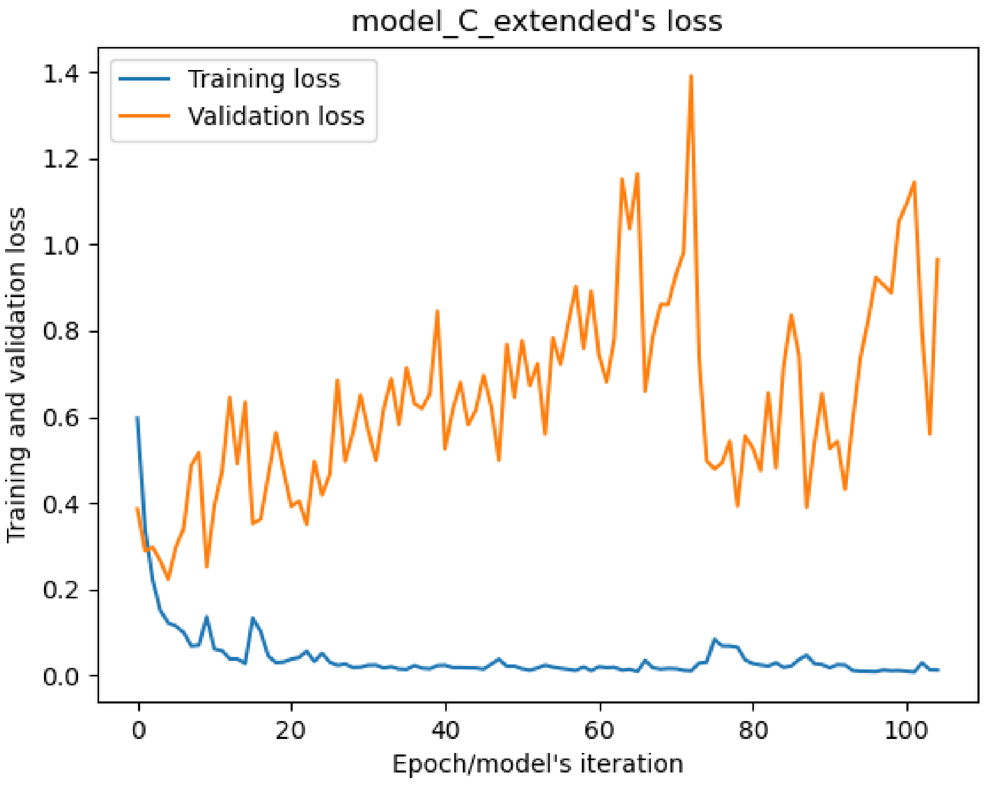


Trained with the extended ransomware dataset

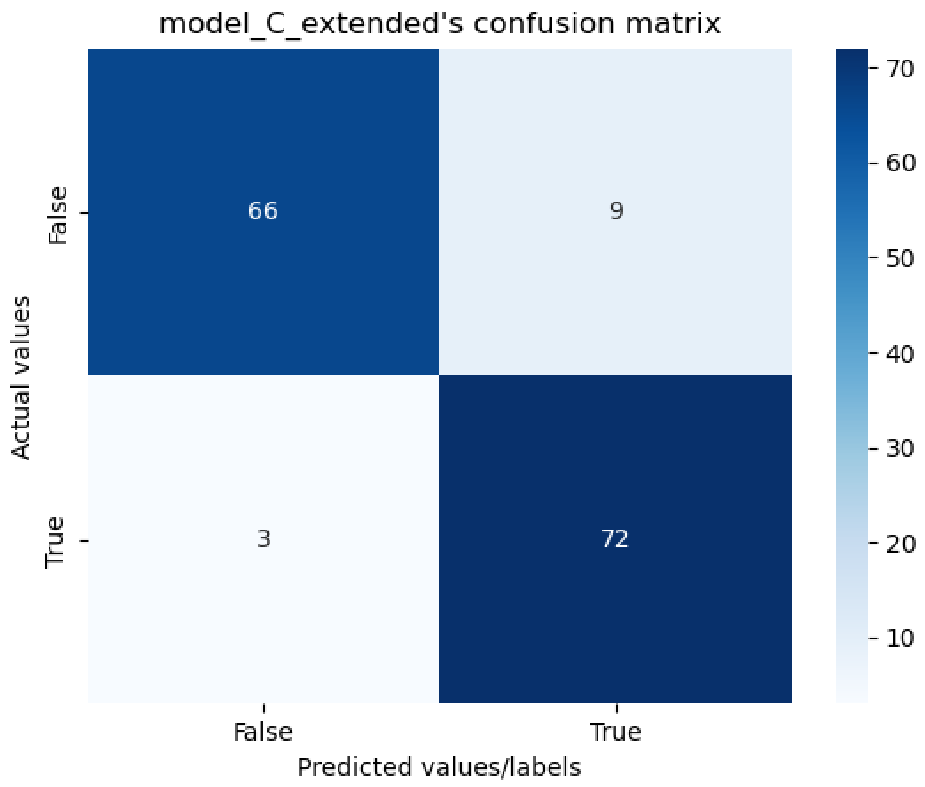
Graph of the model C (trained on extended ransomware dataset)'s accuracy values during the training process:



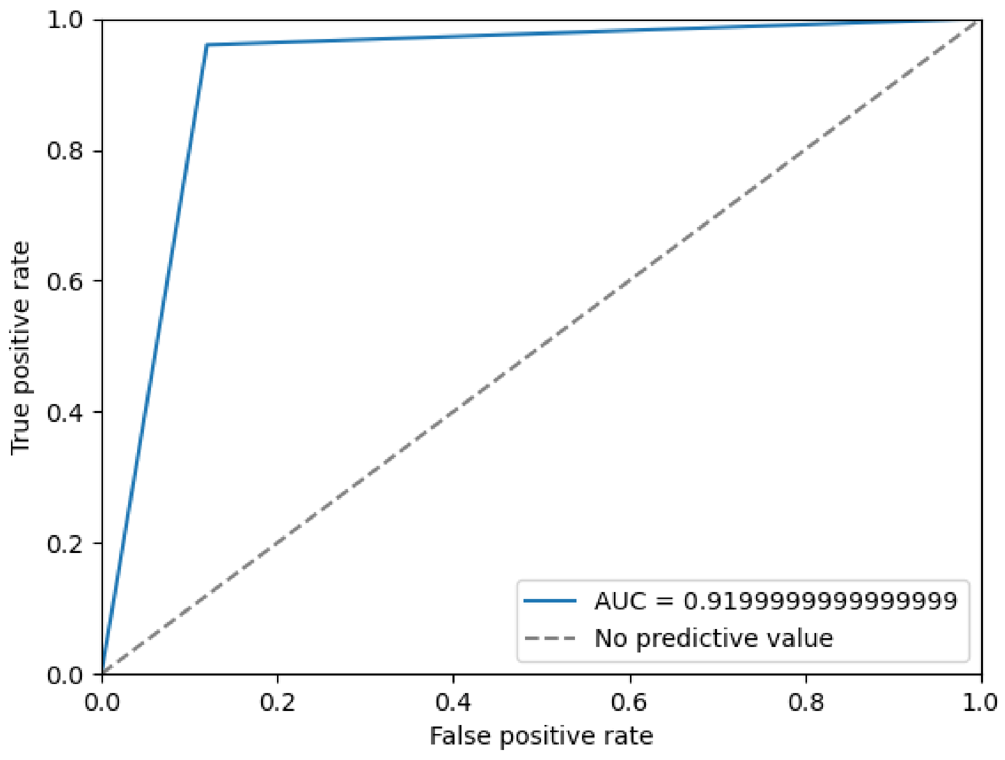
Graph of the model C (trained on extended ransomware dataset)’s loss values during the training process:



Graph of the model C (trained on extended ransomware dataset)'s confusion matrix:



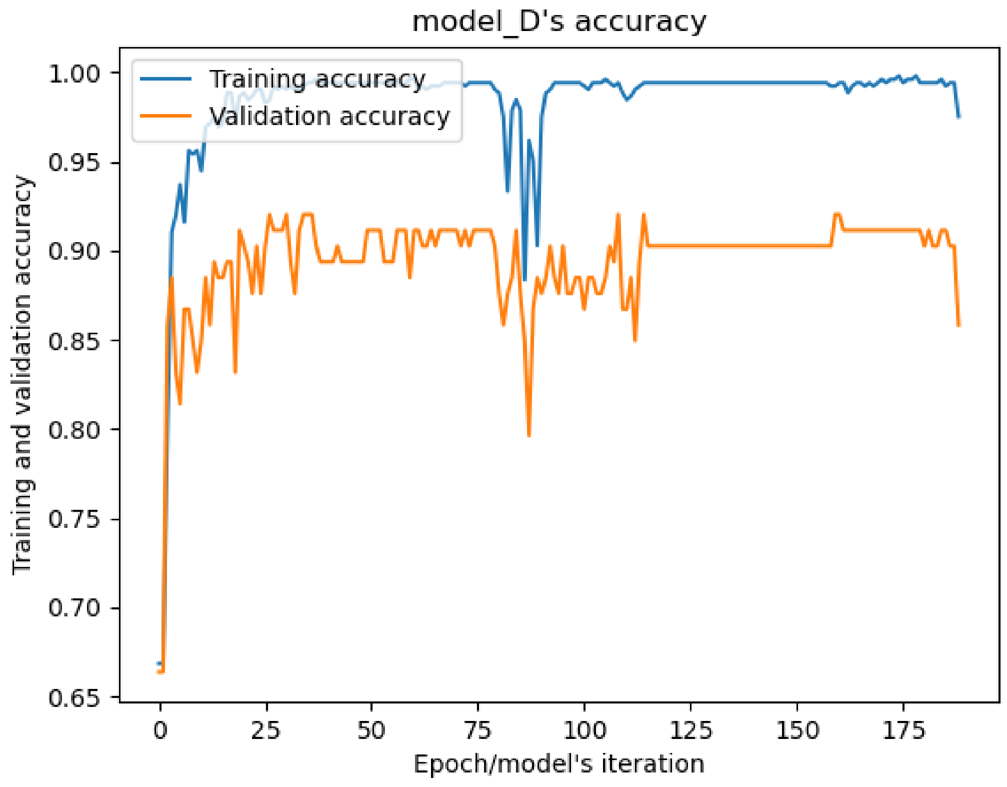
Graph of the model C (trained on extended ransomware dataset)'s ROC curve:



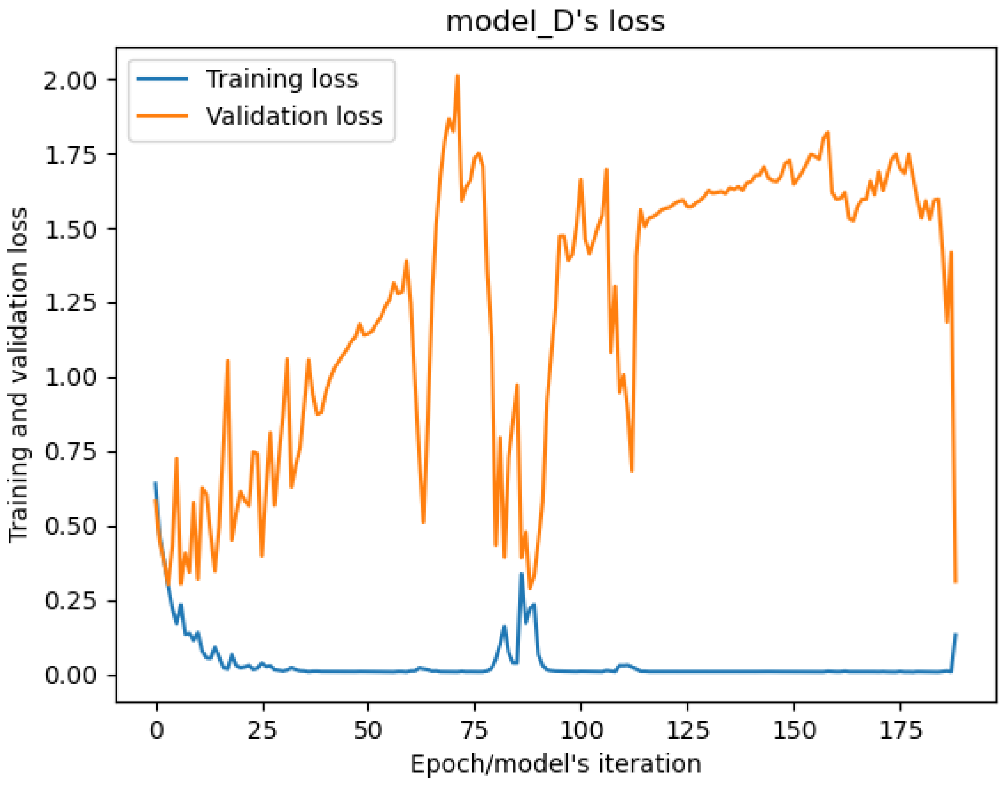
Model D's plots

Trained with the initial ransomware dataset

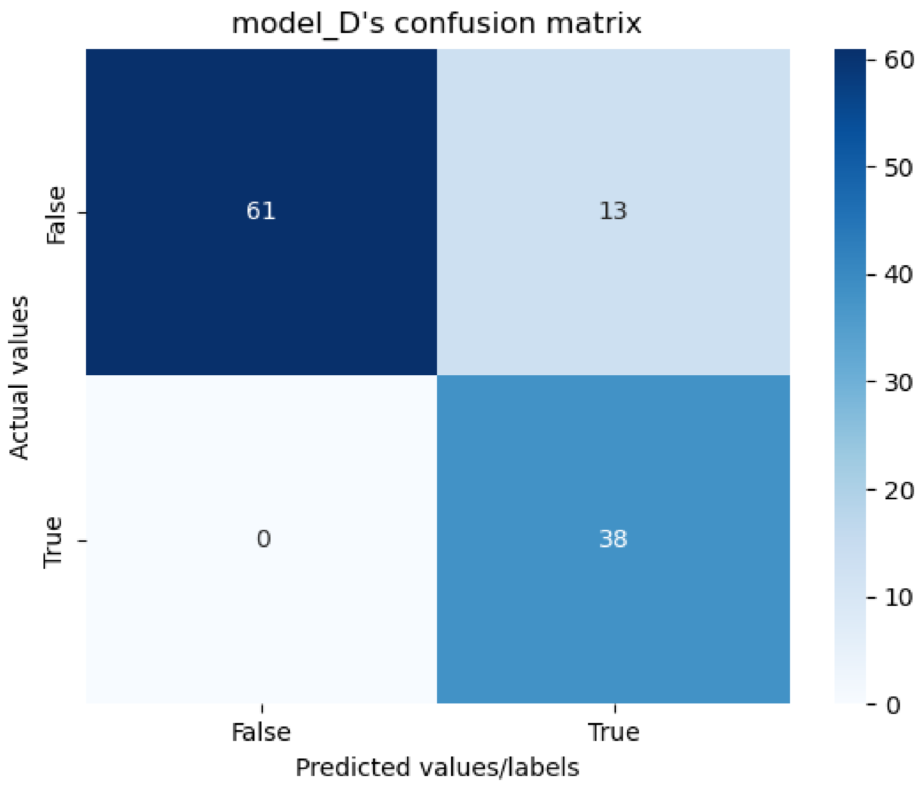
Graph of the model D (trained on initial ransomware dataset)'s accuracy values during the training process:



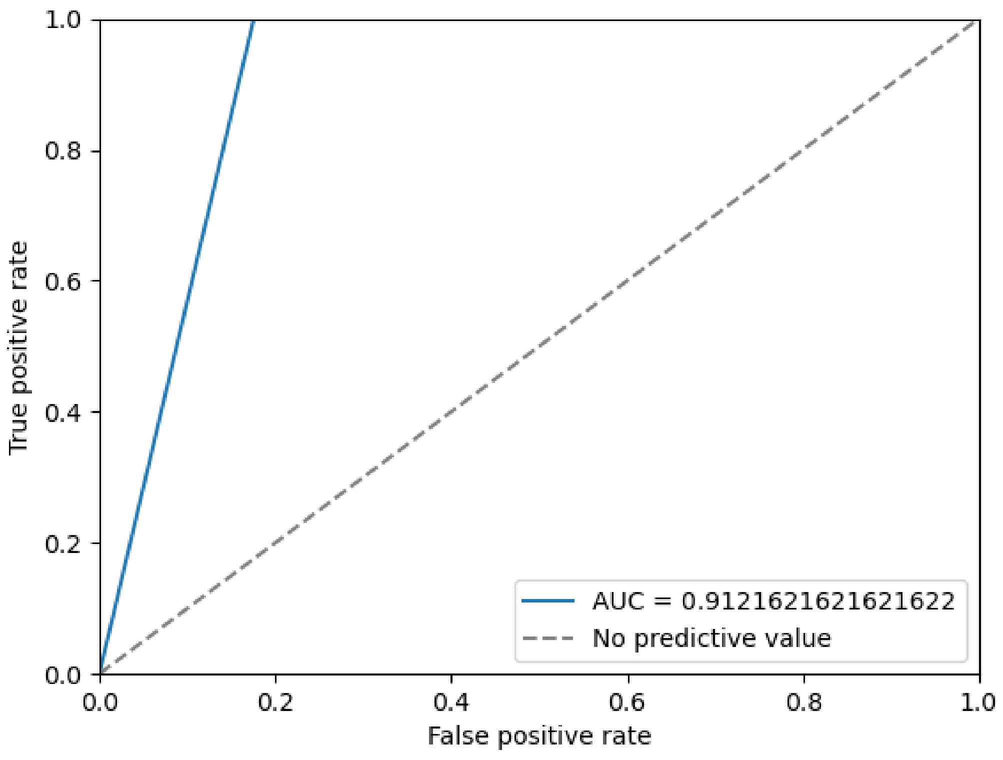
Graph of the model D (trained on initial ransomware dataset)'s loss values during the training process:



Graph of the model D (trained on initial ransomware dataset)'s confusion matrix:

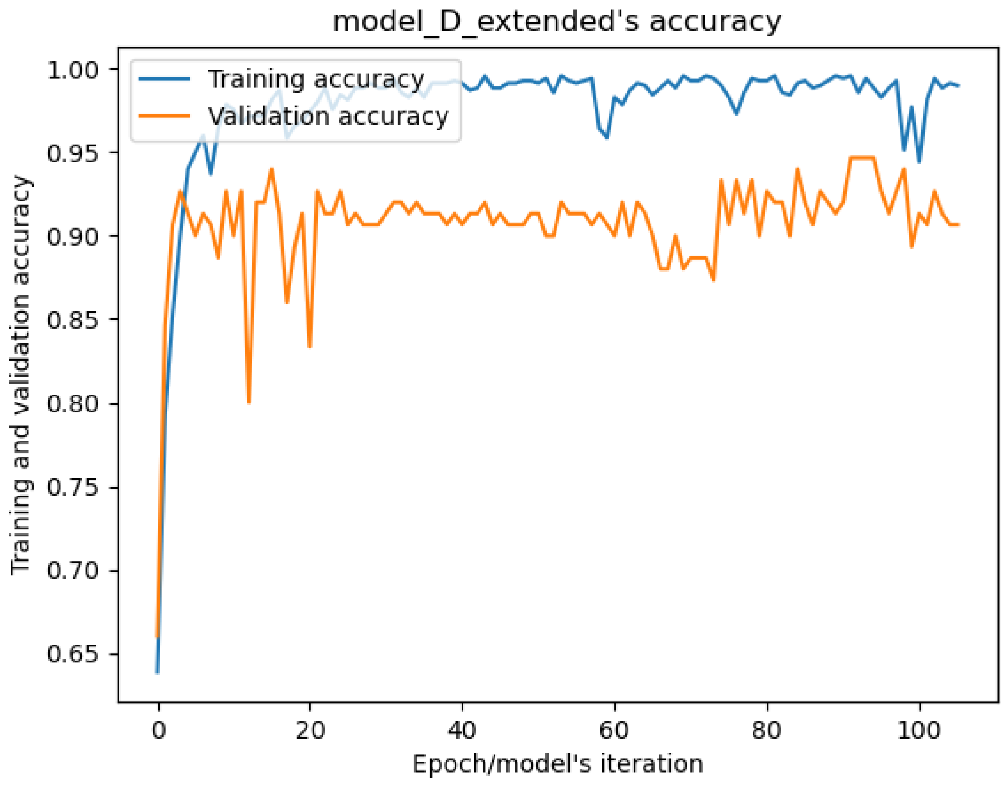


Graph of the model D (trained on initial ransomware dataset)'s ROC curve:

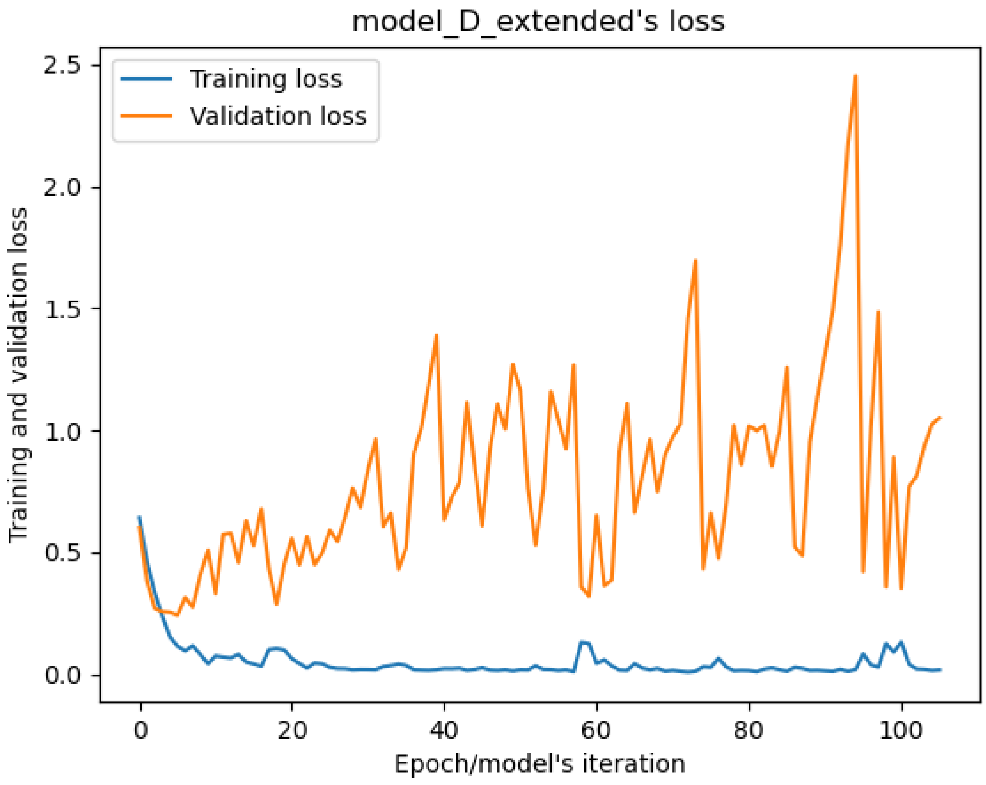


Trained with the extended ransomware dataset

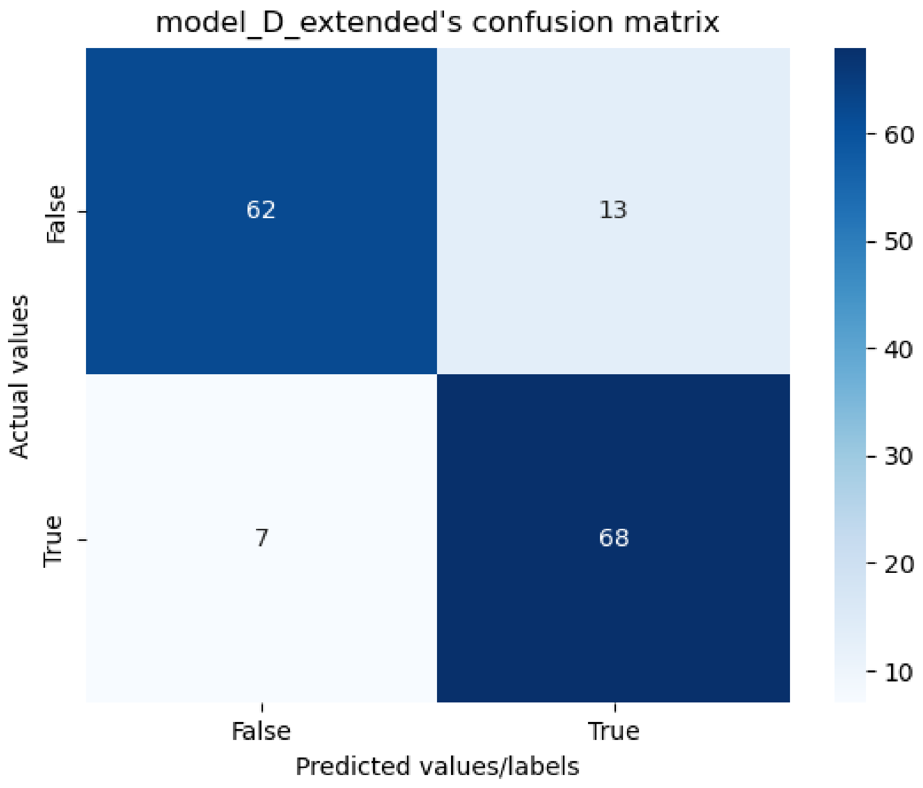
Graph of the model D (trained on extended ransomware dataset)'s accuracy values during the training process:



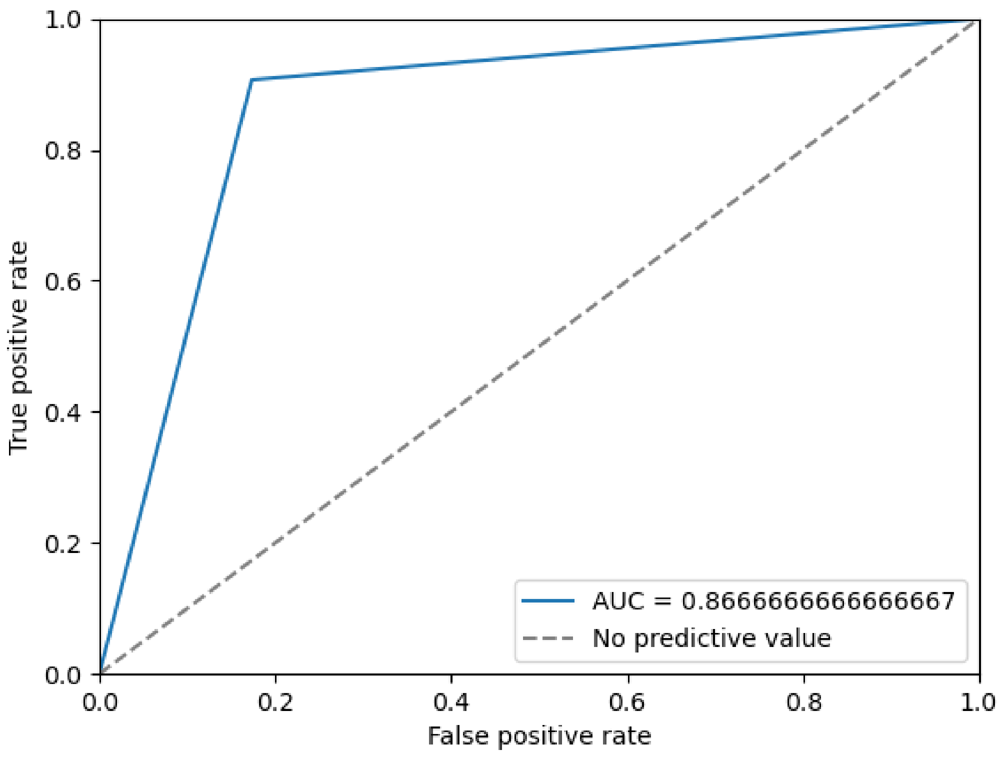
Graph of the model D (trained on extended ransomware dataset)'s loss values during the training process:



Graph of the model D (trained on extended ransomware dataset)'s confusion matrix:



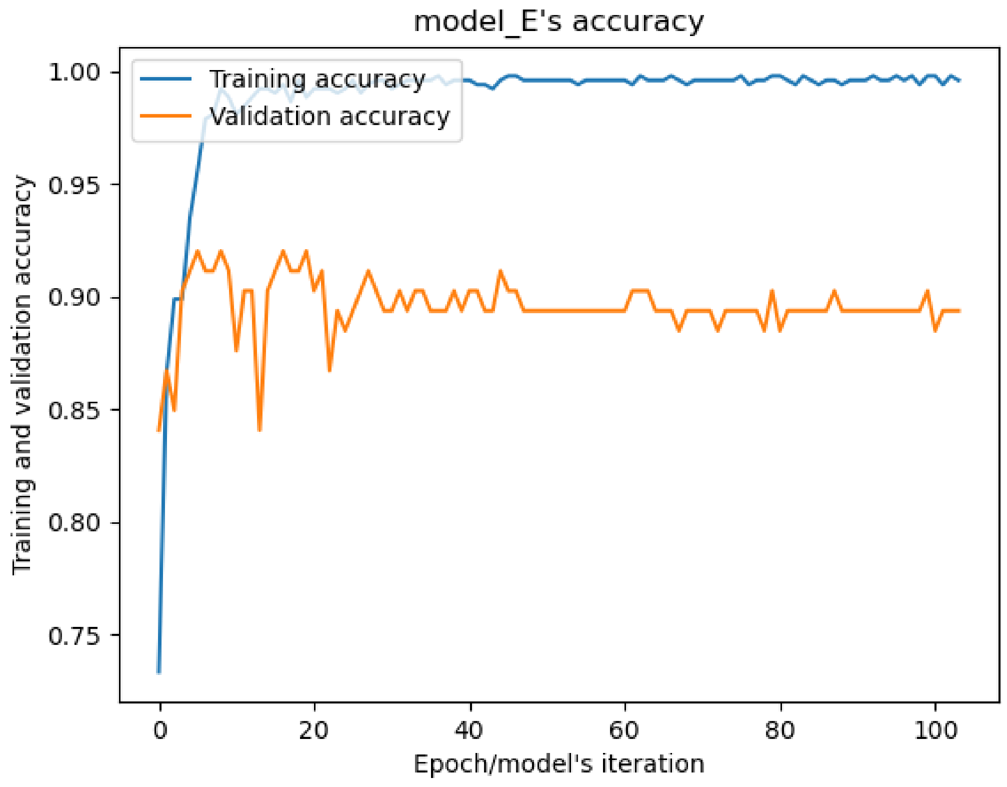
Graph of the model D (trained on extended ransomware dataset)'s ROC curve:



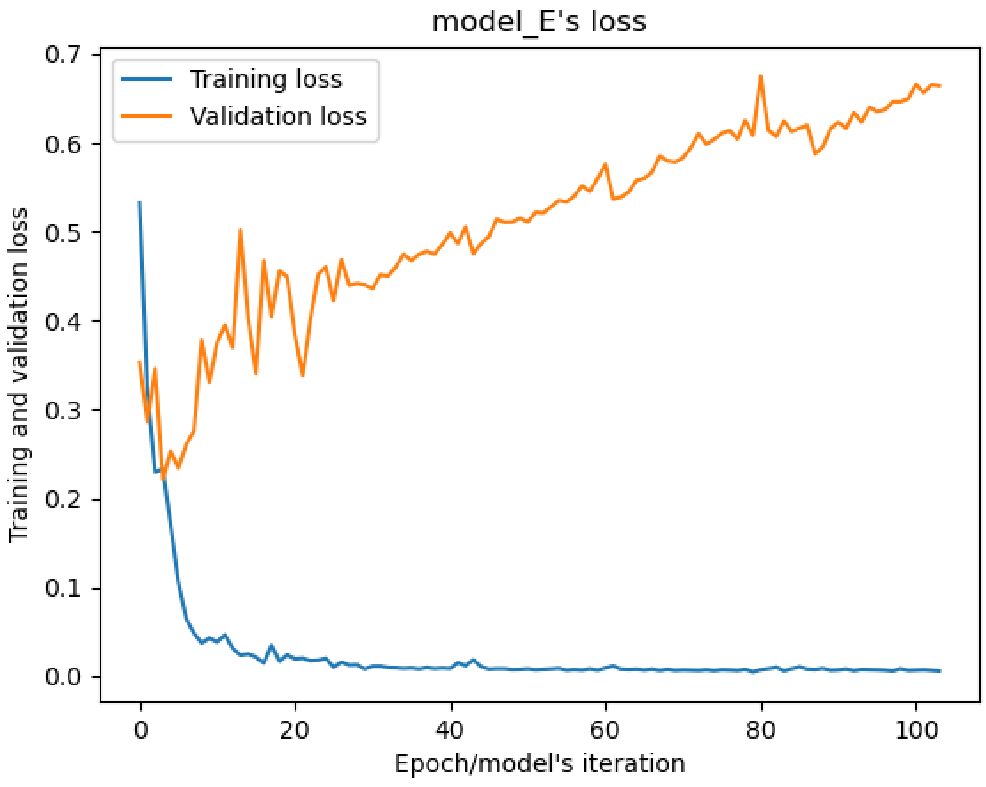
Model E's plots

Trained with the initial ransomware dataset

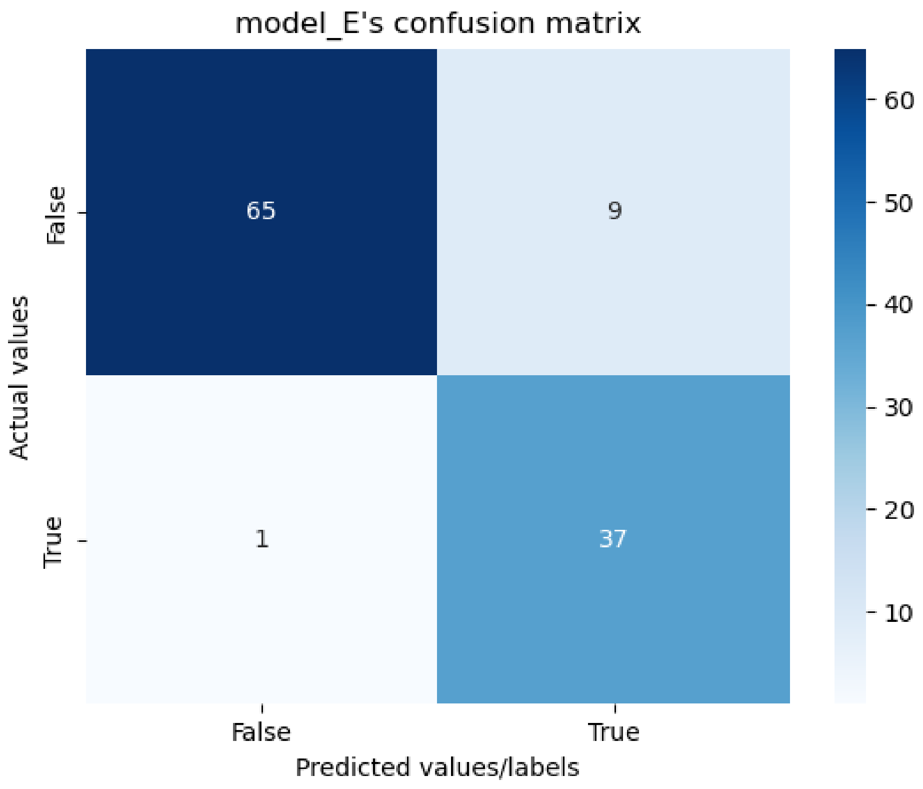
Graph of the model E (trained on initial ransomware dataset)'s accuracy values during the training process:



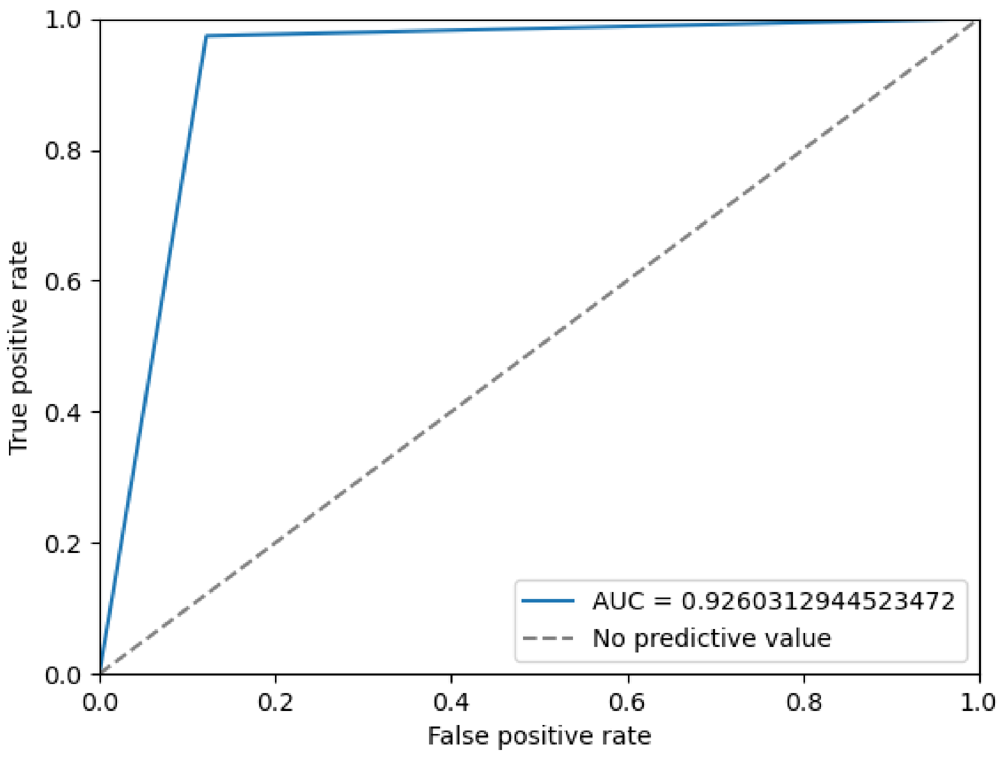
Graph of the model E (trained on initial ransomware dataset)'s loss values during the training process:



Graph of the model E (trained on initial ransomware dataset)'s confusion matrix:

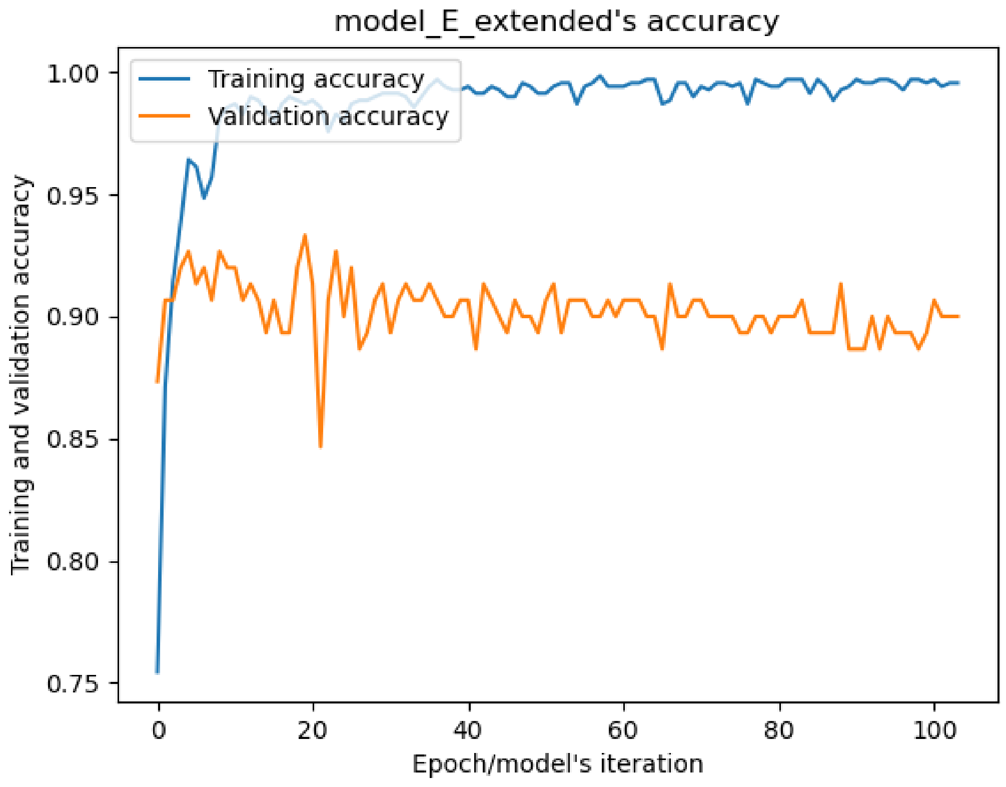


Graph of the model E (trained on initial ransomware dataset)'s ROC curve:

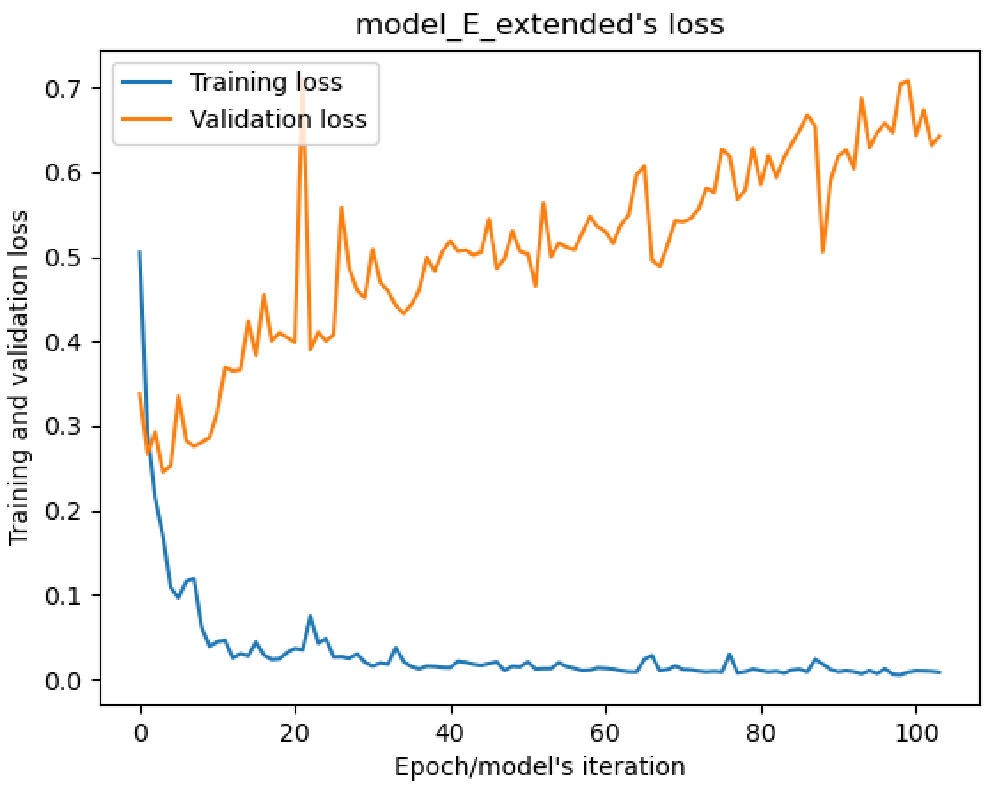


Trained with the extended ransomware dataset

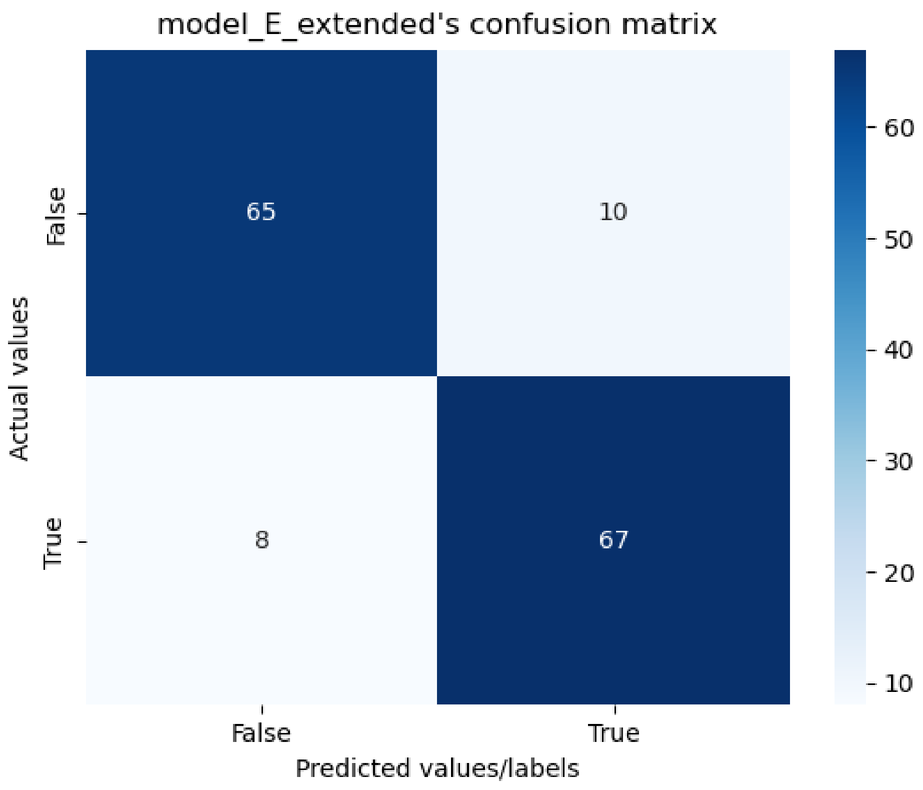
Graph of the model E (trained on extended ransomware dataset)'s accuracy values during the training process:



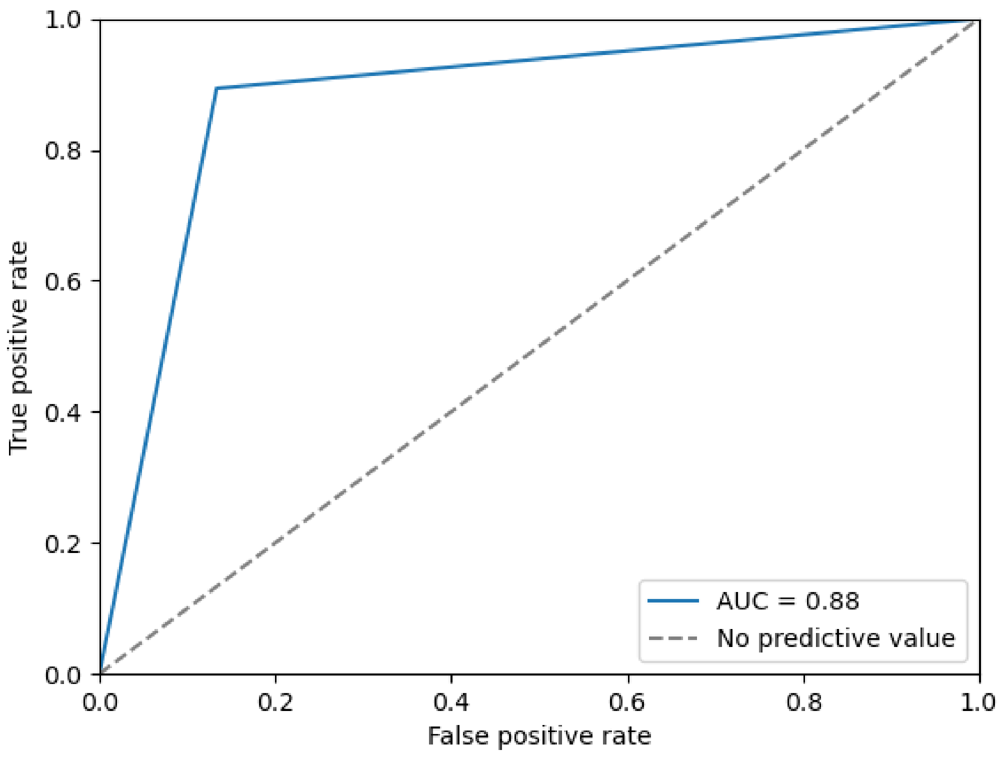
Graph of the model E (trained on extended ransomware dataset)'s loss values during the training process:



Graph of the model E (trained on extended ransomware dataset)'s confusion matrix:



Graph of the model E (trained on extended ransomware dataset)'s ROC curve:


True positive rate
False positive rate
AUC = 0.88
No predictive value