Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

AICryptojackingTrap
A Machine Learning-Based Approach for Cryptojacking Detection

Supervisor
Prof. Marco Roveri
Co-supervisor
Dott. Atefeh Zareh Chahoki

Student
Lorenzo Masè

Academic year 2023/2024

Contents

Summary

Cryptojacking is a form of malware attack where cybercriminals exploit a victim's computing resources to mine cryptocurrency without permission. This malicious activity, which occurs covertly in the background, is highly profitable for attackers, but poses serious issues for victims, such as slowed device performance, reduced hardware lifespan, and increased energy costs. Due to the profitability of cryptocurrency mining, cryptojacking has become increasingly attractive to cybercriminals. Although there are many cryptojacking detection methods, attackers often find ways to bypass these defences by continuously evolving their evasion techniques.

To address these challenges, CryptojackingTrap was developed as a robust cryptojacking detection tool. It operates by identifying hash function data which is required in all cryptojacking activities, even those disguised to avoid detection. CryptojackingTrap uses assembly-level analysis to track memory access patterns in suspicious executables and correlates these data with cryptocurrency peer-to-peer network activity. This approach is resilient to the most known criminal evasion techniques, but it needs considerable processing for detection that can be improved and made more efficient.

This thesis introduces AICryptojackingTrap, which is built upon the infrastructure of CryptojackingTrap, taking advantage of its evasion resilience while aiming to reduce computational overhead through the application of machine learning techniques in its detection module. This approach makes AICryptojackingTrap more practical for widespread use. The research involved constructing a customised data set from memory read log files, preprocessing these data with normalisation techniques, and training a machine learning model to identify patterns associated with cryptojacking. Throughout this process, various machine learning techniques were explored to determine the optimal model that maintains accuracy while minimising resource usage.

Experimental results demonstrate that the problem is not trivial and that a complex model is needed to obtain a more robust solution. The model trained did not learn the pattern to recognize cryptojacking, resulting in a 54% accuracy and 69% loss during the training phase. The evaluation results in 0% accuracy on mining data and 100% accuracy on non-mining data.

This research shows how there is potential for a more complex solution to the problem; future works aim to develop the study with a more in-depth analysis of memory-read log files and the amount of relevant information they contain.

1 Introduction

Cybersecurity is an important area within Computer Science, as new technologies often lead to security challenges. One of the most serious threats in recent years is cryptojacking. This section defines cryptojacking attacks and explains why they pose a significant risk to users when browsing the web or downloading applications. Additionally, it describes the purpose of this thesis, which is to focus on detecting this type of malware.

Mining is the process in which a user uses the resources of his computer to perform complex calculations to create a new block on the blockchain. Once the block is successfully mined, the user receives cryptocurrency as a reward. Bitcoin and Litecoin consensus algorithm is called proof of work (PoW), while Ethereum uses proof of stack (PoS). The calculations made by the users in some consensus algorithms, including PoW, are highly CPU intensive, which may affect the execution time for other operations during the mining process. These consensus algorithms are mathematical puzzles that miners need to solve using their computing resources to receive newly minted cryptocurrencies and a fee as a profit once it is solved [18].

In the last decade, mining cryptocurrencies has become extremely popular because of their high profits, and this led malicious users to be interested in exploiting mining to monetise their victim's resources more efficiently. For this purpose, cybercriminals would explore methods to exploit the victim's resources to mine cryptocurrencies on their behalf without the victim's knowledge and agreement, this malware process is called cryptojacking, coinjacking, or drive-by mining [11]. Cybercriminals would use two different ways to execute the mining algorithm, either with browser-code execution or with an executable file downloaded by the victim masked as a movie, a video game, or music [23]. The impact of cryptojacking is very high since we can identify a breach in availability, therefore the victim would notice an important slowdown of the machine, the electricity costs would also reflect on the victim, and the hardware lifespan is reduced [5].

The 2023 SonicWall Cyber Threat Report provides us with data on the impact of cryptojacking in the world as it states:"In North America, which typically sees by far the most attacks, volume rose from 78.0 million in 2021 to 105.9 million in 2022 — a 36% increase and more than the world saw the previous year. Asia saw an even larger year-over-year increase of 129%, jumping from 3 million to 6.9 million. But it was Europe where cryptojacking grew the fastest: The volume there soared from 3.4 million in 2021 to 22.0 million in 2022, an increase of 548%. Despite the sky-rocketing attacks in Europe, the United States remained the country with the highest volume. Cryptojacking attempts there rose 41% year over year" [25]. During 2022 there was a peak of cryptojacking attacks, and now they are slightly reducing in some parts of the world. This information comes from the latest publication from SoincWall, 2024 SonicWall Mid-Year Cyber Threat Report that cites "After a record-breaking year, Cryptojacking dropped 60%. Most of the world saw a decrease, except India, which saw a staggering 409% increase" [26]. Following the presented statistics, cryptojacking is one of the most critical threats to the cybersecurity field, and it has a major impact on the secure fruition of online services that need to be thoroughly addressed in the security domain.

This thesis proposes a novel machine learning approach for cryptojacking detection, expanding on the foundational infrastructure established by CryptojackingTrap. Using insights from CryptojackingTrap and other detection methods, this study advances detection efficiency and computational practicality. An initial analysis of the data set derived from CryptojackingTrap examines memory-read log files and their applicability to detect cryptojacking activities. Building on this, a new dataset is constructed specifically to train the model, incorporating custom encoding algorithms and file modification techniques using regular expressions. After evaluating multiple encoding strategies, the study identifies a single optimal technique, facilitating a higher detection rate. Additionally, a preprocessing algorithm is implemented to address format inconsistencies within the memory-read logs, followed by a normalisation algorithm tailored to enhance machine learning performance. The final model, after extensive training and validation, reflects the potential of the proposal to improve cryptojacking detection. However, the results indicate that further development is necessary to achieve more comprehensive detection capabilities. This underscores the complexity of detecting cryptojacking within diverse log data using machine learning and the need for continued refinement and advanced analysis techniques to fully address the problem.

The organisation of this paper is as follows. The background section (2) discusses typical methods for detecting cryptojacking, particularly exploring the details of the CryptojackingTrap [4] approach, which serves as the basis for improvement in this study. In the AICryptojackingTrap solution section (3), the main solution of this thesis is presented, detailing the technologies employed to achieve the results and the specific task addressed, outlines how the initial dataset was analyzed and pre-processed to align with the goals of this thesis, and is explained an analysis of the possible algorithms to use for the model. The experimental evaluation section (4) describes the environment in which the algorithms were executed, the starting dataset, and the algorithms used for the proposed solution. This chapter also includes the implementation results and evaluation outcomes. Finally, the conclusions and future work section (5) provides an overview of the work completed and suggests potential further improvement.

2 Background

This section explores various methods for detecting cryptojacking, specifically in-browser and executable-based attacks, using static and dynamic analysis techniques, as well as machine learning approaches. It delves into the CryptojackingTrap framework [4], analyzing its methodology and performance as the foundation for this thesis development. The rationale behind selecting CryptojackingTrap as the basis for this work is discussed in detail, emphasizing its resilience to common evasion techniques and its potential for improvement through machine learning integration.

2.1 Detection techniques

Cryptojacking's research, as surveyed in [18] can be differentiated into two main fields: "Detection approaches" and "Analysis techniques and their features". The detection approaches consist of conventional approaches such as signature-based solutions as well as anomaly detection solutions which are based on machine learning or deep learning. "Analysis techniques and their features" can be studied under two major categories static and dynamic analysis. Static and dynamic analysis approaches use different features extracted from code, network, browser, hash, and hardware. The next sections will explain both detection approaches and analysis techniques. 2.1 illustrates the current cryptojacking research taxonomy.

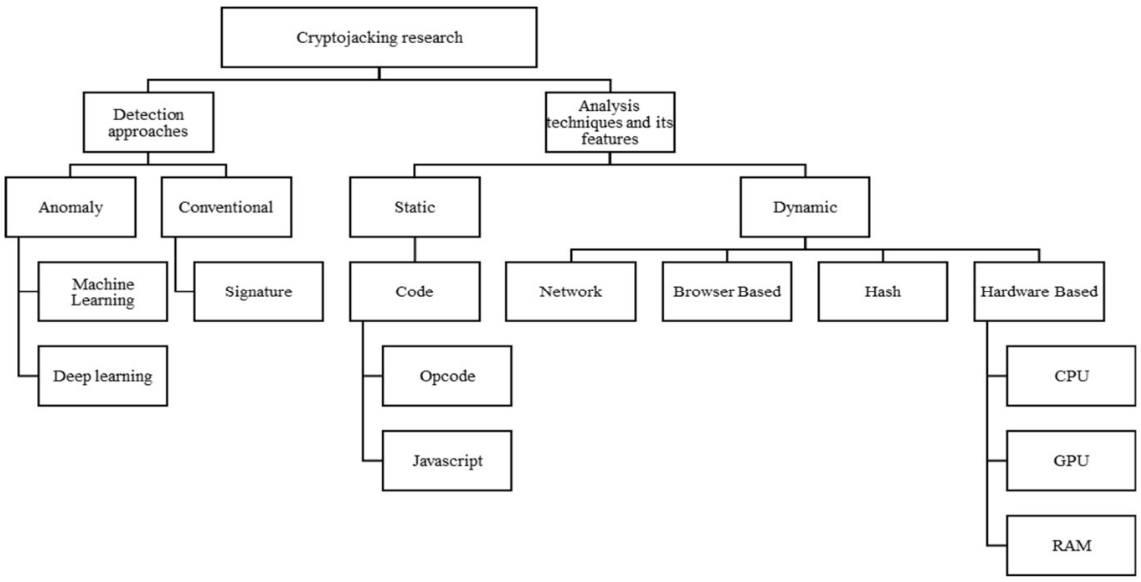


Figure 2.1: Taxonomy of cryptojacking [18]

In the remainder of this section static analysis, dynamic analysis, and machine learning approaches are discussed.

2.1.1 Static analysis

Static analysis is a well-known technique used to check whether a file is malicious or contains vulnerabilities. Static analysis is based on reverse engineering a target executable to obtain its source code and analyze it. In the case of cryptojacking, static analysis can be used to check the code of software if any suspicious operation is being used, such as specific constant values that are used in cryptojacking hash functions or any code and address for mining pools.

In this detection category, the goal is to check for any code that could start a mining bot or establish a connection with a bot master. A bot master can be defined as: "The commander and controller of a botnet. They remotely direct compromised computers to execute various tasks, often for malicious purposes" [29]. Static analysis is used to check the flow of the program, which are all the possibilities that can occur and lead the program to unwanted behavior. 2.1 shows how static analysis has been used mainly with opcodes or JavaScript.

Static analysis is one of the well-known approaches for security analysis purposes, consequently, it is also known by cyber-criminals to explore and develop its evasion mechanisms. Static analysis can be evaded by using code obfuscation resulting in inefficiency in static analysis [22]. Although static analysis remains one of the most effective detection methods, it suffers from certain limitations, such as its inability to detect sophisticated or polymorphic malware, making it unreliable in some cases.

2.1.2 Dynamic analysis

Dynamic analysis is a technique based on analyzing something that is in the process of execution. In the case of software, it is applied during the execution of the software itself inside a controlled environment and checks features during different phases of the software.

The features studied in [18] are network, browser, hash, and hardware features (CPU, GPU, and RAM), discussed in the following:

- **Network:** The network data is invaluable for detection purposes since it could provide us with information about a connection established with a bot master or mining pools directly depending on the botnet architecture.
- **CPU:** Since cryptojacking is a resource-intensive activity that places a significant load on the CPU, an unexplained increase in CPU usage—without any user-intended processes running—is often one of the first indicators that an unauthorized process may be executing in the background.
- **Browser:** The browser could contain some malicious code for starting the mining process such as HTML or JavaScript code. An analysis of the browser could then lead to an understanding if a web application is malicious or not.
- **Hash:** It is possible to check how many times, or how much time is spent to execute hash functions by an application. Otherwise, it is possible to check the hash function used by the application, this could contain relevant information for the detection.
- **RAM:** The memory could contain some suspicious traces of mining. An example is the CryptojackingTrap work, in which it is possible to detect mining by checking the payloads retrieved by applications and checking them with an algorithm.

Dynamic analysis is another widely used technique for detecting cryptojacking, but it has become less reliable as cyber-criminals have adapted their methods to evade detection. Various strategies can be employed to bypass dynamic analysis, the next list contains the evasion techniques used to evade the features mentioned previously in order:

- **Connection encryption:** Encryption would be the issue related to traffic analysis. Encrypted connections would not be analyzable. Nowadays, bots can easily set up encrypted connections with the bot master.
- **Delayed cryptojacking:** The usage of the CPU can be manipulated by cyber-criminals by reducing the mining rate or starting mining only after some action made by the user. These actions could be: starting a video, opening a game, or starting a song. This kind of attack has been called delayed cryptojacking [31]. This technique is used to evade, CPU usage detection, and memory read detection, as the application would retrieve less significant information in the same amount of time.
- **Code obfuscation:** This technique is based on hiding malicious code so that the analysis does not detect any strange behavior from the application. This technique is used to evade detection based on hash, and browser analysis.

- **Coded hash functions:** The dynamic analysis of hash functions can be evaded from cyber-criminals by creating their hash functions which would become very hard to analyze. It is needed for the analysis that the bot uses OS APIs for calculations [20].
- **Low-rate mining:** Low-rate mining is a technique used by cyber-criminals that reduces the number of calculations made by the bot so that the detection is less likely. This kind of technique can evade both CPU usage detection and memory read detection.

The study in [31] introduces a novel attack technique known as Delay-CJ. This research explores how delayed cryptojacking attacks are particularly challenging to detect, as they balance the cyber-criminal's trade-off between lost mining revenue when a user is actively browsing and a reduced chance of detection by analysts. By initiating mining activity only after the user has executed a specific action such as starting a video, or pressing a button. These delayed attacks lower the probability of being noticed during active monitoring.

Consequently, although dynamic analysis is a powerful detection method, cybercriminals continue to adapt at the pace of advancing technology, rendering many techniques ineffective at reliably identifying cryptojacking activities

2.1.3 Machine learning

Machine learning is a technique that is becoming more and more popular in intrusion detection in comparison to static and dynamic analysis, as shown in studies that are conducted in [23]. Many solutions proposed with machine learning have had accurate results using online datasets. These works are based on training a machine-learning model or a neural network to classify the suspicious data for mining and not mining categories by using different features. How the models learn is based on what algorithm, and what kind of model it decides to use. In machine learning detection, researchers mostly use decision trees, random forests, naïve Bayes models, support vector machines (SVM), or K-nearest neighbors algorithms (KNN). The models are learning the behavior of different features in some given situations to be able to distinguish between mining and non-mining data entries. The recent research mainly used information on CPU, network traffic, and memory as their features.

Deep learning is a machine learning method based on deep neural networks, it is possible to find convolutional neural networks (CNN), long short-term memory networks (LSTM), and recurrent neural networks (RNNs). These work differently from the machine learning models as these are based on layers and neurons that are computing some input values to predict an output.

Both machine learning and deep learning methods can be used to propose different models that have different advantages and disadvantages which are discussed in Section 3.6.

One of the disadvantages of machine learning is not enough datasets and information related to cryptojacking [23]. Different models may use the same type of datasets leading to possibly similar results. This makes training the model very hard since in machine learning datasets need to be very large and with heterogeneous data. Otherwise, the model would not learn as well as it is expected and could have low accuracy in processing the real-world input.

The research is still evolving mainly trying to resolve problems related to the detection of fileless malware within a system. Other papers are focused on finding new methods of resolution, new features, new usable code, or new datasets that could be used in the future [18].

As the research progresses, machine learning detection is becoming one of the most adapted techniques to detect cryptojacking. Concerning the malware evasion aspect, if a model is correctly trained, it could be resilient to evasion techniques such as delayed attacks and low-rate mining. This would be possible if the model was given as input different data related to these kinds of attacks, leading to understanding them.

2.2 CryptojackingTrap

This thesis is based on the malware detection approach introduced by BotcoinTrap [32] and expanded as CryptojackingTrap [4]. CryptojackingTrap is a solution inspired by the Venus Flytrap (Dionaea muscipula) plant since its detection algorithm works with similar time windows and heuristics. CryptojackingTrap has two major sections, the core and the extensions. The core includes the monitor and the detector. The extensions are blockchain-specific listeners, each one has a link to that specific blockchain network to retrieve information needed for the detection.

The monitor module tracks a process running on the host for a time frame, traces all its memory reads, and outputs a log file containing all the memory read content retrieved and their timestamps. This memory-read log file contains every access made by the suspicious software from the memory. It is implemented using `Pin` debugger: "Pin is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures that enables the creation of dynamic program analysis tools. Some tools built with Pin are Intel® VTune™ Amplifier, Intel® Inspector, Intel® Advisor, and Intel® Software Development Emulator (Intel® SDE). The tools created using Pin, called Pintools, can be used to perform program analysis on user space applications on Linux*, Windows* and macOS*. As a dynamic binary instrumentation tool, instrumentation is performed at run time on the compiled binary files. Thus, it requires no recompiling of source code and can support instrumenting programs that dynamically generate code" [17].

Each extension module monitors a specific blockchain network, logging critical data for use in the detector module. In CryptojackingTrap's current setup, this logged data includes the block hash and the timestamp of each block's creation. The block hash is particularly essential, as mining operations require the latest block hash as an input. Consequently, the hash of the most recent block becomes a predictable value, which is indispensable for initiating mining activities.

The solution to detecting cryptojacking activity involves monitoring whether this predictable block hash value is ever accessed from memory by the suspicious program. A key challenge in this approach arises because the hash value is often divided into multiple segments within the memory-read log file, making the detection algorithm more complex. To address this, the detector module applies the algorithm to examine memory reads against the hash value. It checks if any segments of the latest block's hash are retrieved from memory within fixed time intervals in a significant repetition, thereby indicating potential mining activity.

To improve the accuracy of the detection CryptojackingTrap's detector module algorithm operates across multiple levels of abstraction, each designed to detect a specific level of suspicious activity detection. The layers consist of split occurrences, hash occurrences, mining occurrences, and CryptojackingTrap occurrences. At each level, the algorithm increases the criteria and narrows down the detection results. This layered detection strategy significantly minimizes false positives and false negatives, achieving near-zero error rates. Evaluations conducted with benign miners, randomized memory-read files, and non-mining applications confirm the algorithm's precision and adaptability across diverse scenarios.

CryptojackingTrap presents a novel approach to detecting cryptojacking by moving beyond traditional methods that typically focus on high-level analyses of CPU usage or network traffic. Instead, it examines assembly-level instructions to gain a more detailed understanding of what processes are doing on a victim's machine, offering a clearer distinction between legitimate and malicious activities. One of the key strengths of CryptojackingTrap is its resilience to evasion tactics. Because it relies on low-level instructions and the use of predictable fields that are necessary for mining, it makes it much more challenging for cybercriminals to avoid detection. This solution represents a significant step forward in detecting cryptojacking with a high level of accuracy and resilience.

While CryptojackingTrap presents a resilient approach to detecting cryptojacking through assembly-level analysis, this thesis aims to explore potential improvements in its detector module performance by incorporating machine learning techniques. Although CryptojackingTrap's algorithm is highly evasion resilient, its computational cost remains significant, particularly when processing large memory-read log files. The goal of this thesis is to investigate how machine learning could be used to optimize the detection process, potentially reducing process time and computational overhead while maintaining the algorithm's strong evasion resilience. This exploration seeks to enhance the overall efficiency of the detection module, aiming for a more scalable and efficient solution.

3 AICryptojackingTrap solution

This chapter details the development of the machine learning model built on the foundation of CryptojackingTrap. It covers the technologies employed in the design and implementation, outlines the task definition and the operations made on the starting data set to create a new dataset, and provides an analysis of the most commonly used algorithms to address the cryptojacking detection problem to decide which one to use.

3.1 Development setup

This section outlines the tools and resources used to develop the machine learning model, including the programming languages and libraries critical for data science and machine learning. In the remainder of this chapter, the libraries used to implement the desired functionalities in each function are described.

Python is used to develop the code for this model, the preprocessing scripts, the normalisation, training, and evaluation algorithms. Different libraries are used in this development to facilitate the work with a significant amount of data.

The libraries used in this development are as follows.

- `csv`: "The CSV module implements classes to read and write tabular data in CSV format" [9]. This library is used to create the CSV file from the initial dataset provided by CryptojackingTrap.

- `re`: "This module provides regular expression matching operations. Both patterns and strings to be searched can be Unicode strings (str) and 8-bit strings (bytes)" [10]. This library is used to define correct expressions inside the log files as a pre-processing phase.

- `numpy`: "NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation, and much more" [8]. NumPy arrays are used in this work to store data and then used to train the model.

- `pandas`: "Python library used for working with datasets. It has functions for analysing, cleaning, exploring and manipulating data" [30]. Used to import, modify, and extract data from different files.

- `scikit-learn`: "Scikit-learn is an open-source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection, model evaluation, and many other utilities" [24]. This library was used to split the dataset using `sklearn.model_selection`'s `train_test_split` function, which splits arrays and matrices based on a relation between the sizes wanted for the training set and the testing set.

- `tensorflow`: "This open-source machine learning platform supports you in efficiently building machine learning models. TensorFlow assists with all stages of the process, from data preparation to model running. TensorFlow gives you access to several tools and libraries for both machine learning and deep learning in several languages" [6]. It is used to create, train, save, and evaluate the model using `keras` as a higher-level library. Keras is a high-level interface developed in Python that runs with Tensorflow, it is a framework that makes the definition and creation of models easy, but can be used on different levels of complexity [19]. The functions used are the following:

- `tensoflow.keras.models.Sequential()`: Constructs the model as a sequential stack of layers [3].
- `tensorflow.keras.layers.Dense()`: Used to define the dense layers used in the neural network [1].
- `model.compile()`: Configure the metrics and losses of the model [2].
- `model.fit()`: Trains the model on a dataset, iterates through the dataset based on the number of epochs [2].
- `model.evaluate()`: This function evaluates the model on a new set of data that is not used for training to evaluate the accuracy of the model in the real world [2].
- `model.save()`: Saves the trained model [28].
- `models.load_model()`: Loads a trained model using its path that was saved with `model.save()` [27].

3.2 Task

The task in machine learning is the problem that has to be solved by the model, tasks are defined based on the output that is expected from the model, hence what problem it is needed to solve. Different kinds of task exist:

- **Classification:** The goal in classification is to make the model learn the differences between two objects.
- **Regression:** The model learns how to predict some function and predicts the output y given an input x.
- **Clustering:** The model learns to group different objects with other objects that have similar features.

Many different tasks exist and can be solved using a specific type of learning. In this thesis, the task to be solved is classification, as the model has to classify different samples as mining or not. This task has already been accomplished with good results from CryptojackingTrap using a greedy algorithm, and this model should be able to learn from CryptojackingTrap's results a pattern to define mining and solve the computational problems of the greedy algorithm.

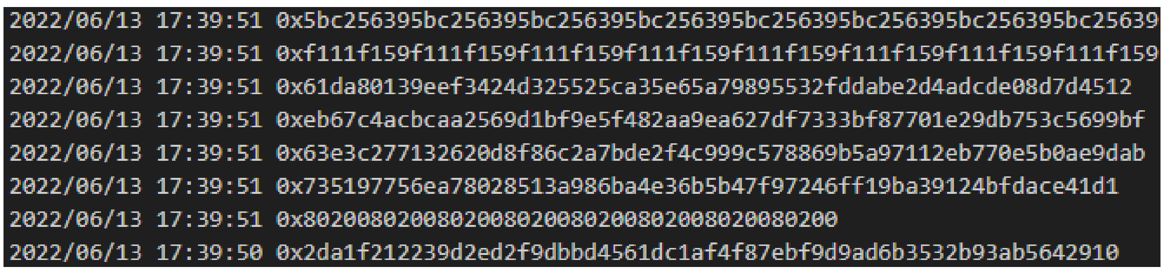
The model during the training phase is expected to create its algorithm to detect the pattern between the memory-read log file and hash value, which defines the differences between mining and not mining. If the string is found multiple times in a fixed time window, it is possible to be sure that the monitored process is mining [4]. Using this idea, the goal is to train a model to have better results and lower inference time than the greedy algorithm developed in the original work. This task is usually solved using eager learner models: these models would create a prediction system during the training phase. This makes the training phase much longer than lazy learning models; these models do not need any training phase, as each time it is given an input, they would compute the whole data set back again plus the new data point. An example of a lazy learner is the K-Nearest-Neighbour algorithm. The task is therefore one of the points that is used to understand what model to use and how to focus its training.

3.3 Initial dataset structure

Memory-read log files are created from the CryptojackingTrap monitor module. These log files contain a date followed by a timestamp for each read and the memory read content retrieved from the application's memory. The image 3.1 shows an example of a read log file.

The starting dataset comprises different memory-read log files and their related hash value for various applications, miner or non-miner. The ideal structure of the memory-read log file is uniform, for each line, it is found a date, timestamp, and memory-read content which is encoded as a hexadecimal number.

The main idea is related to the features to use. The model, to learn the pattern related to mining, has to be able to read the memory-read log file and the hash value. Memory-read log files are critical



```
2022/06/13 17:39:51 0x5bc256395bc256395bc256395bc256395bc256395bc256395bc256395bc25639
2022/06/13 17:39:51 0xf111f159f111f159f111f159f111f159f111f159f111f159f111f159f111f159
2022/06/13 17:39:51 0x61da80139eef3424d325525ca35e65a79895532fddabe2d4adcde08d7d4512
2022/06/13 17:39:51 0xeb67c4acbcaa2569d1bf9e5f482aa9ea627df7333bf87701e29db753c5699bf
2022/06/13 17:39:51 0x63e3c277132620d8f86c2a7bde2f4c999c578869b5a97112eb770e5b0ae9dab
2022/06/13 17:39:51 0x735197756ea78028513a986ba4e36b5b47f97246ff19ba39124bfdace41d1
2022/06/13 17:39:51 0x80200802008020080200802008020080200802008020080200
2022/06/13 17:39:50 0x2da1f212239d2ed2f9dbbd4561dc1af4f87ebf9d9ad6b3532b93ab5642910
```

Figure 3.1: Example of a section of a memory-read log file

big text files, and the hash value is a string. This problem requires an analysis of the task to understand how it is possible to transform a memory-read log file and its hash into a sample.

The goal is to create a new CSV file using the start-memory read log files without losing information and making the data viable for the neural network. The next section describes which algorithm is selected to transform this described initial dataset to a suitable targeted dataset that can be used for training.

3.4 Dataset transformation algorithm

This section describes the process of selecting an appropriate algorithm to transform the initial data set into a suitable training format, ensuring data preparation for the chosen model. The initial data set is extremely large. To start the analysis, it is necessary to understand how memory-read log files can be transformed to a format that is useful for the neural network learning process.

The starting ideas were about how a single memory-read log file could be a single sample. This problem is complicated to solve mainly because the log files have different sizes, and the number of features would be different for each, assuming one feature for memory read content, and to obtain an interesting number of samples, it would be needed to create many different memory read log files manually. We need to understand how to create a fixed number of features for each memory read log file and a method to obtain more samples from a single memory read log file.

It is not possible to decide what to keep or what to eliminate from a human perspective, every memory read content could be important as the hash value can be split, and even those memory read contents that do not contain any hash character could give the model information related to the rate of the hash value appearing in the log file. A possible solution could be to divide a memory-read log file into different parts of fixed size. This would solve both the number of features for each sample and the number of sample problems. This approach was decided not to be used since this method of dividing the memory-read log files would make learning very difficult since the model would try to learn the pattern for very small portions of the log file, which could not contain any significant data.

The idea implemented in the solution is based on a *sliding windows algorithm*. Sliding window problems are related to a fixed length window that is used to efficiently slide through a data structure to check for meaningful subsections of it [12], this algorithm is optimal for our problem since it solves the learning problem related to not significant portions as the model would read many times the same values, making the learning stronger. Furthermore, this approach makes it possible that, for a single memory-read log file, many different samples can be computed.

3.5 Pre-processing

Pre-processing is a very important phase of machine learning in which the goal is to prepare obtained data to train the model and ultimately solve problems. Pre-processing is making adjustments to the data before its main process includes addressing these issues: missing values, not well-formatted data, incompatible encoding, and a non-normalised dataset. The next section explains the necessary adjustments that are required to solve this specific Cryptojacking detection problem to create the final CSV file, which is used to train the model.

Section 3.5.1 describes the algorithm that removes unnecessary data from log files to make them well-formatted and more efficient to process. Section 3.5.2 describes the adjustments regarding the required encoding of the text to have floating numbers that represent characters as features. Section 3.5.3 explores the normalization of the CSV file distribution to obtain better results with the training algorithm.

3.5.1 Data cleaning

Memory-read log files are potentially significantly large files that can affect the performance of the main training process and also include some data that are not useful in the training stage. Here, this algorithm removes these unnecessary data and prepares the optimal well-formatted data to proceed with the training. These files have a predetermined pattern for each line: ”Date Timestamp Memory-read-content”, while just Memory-read-content is useful for the training stage of the proposed algorithm, and other parts of each line should be removed. A Python library is used to recognize this pattern and restructure the file to the target format.

Since this file is generated by a debugger, there are occasionally some errors that also need to be fixed in this stage. One observed error is related to the cases in which two lines are concatenated and end with two above-mentioned patterns of ”Date Timestamp Memory-read-content” in one line. Another error is the cases where the line begins with memory-read-content and the date and timestamp are missing because the `\n` character is mistakenly inserted before memory-read-content in its previous line.

```python
import re
def process_file(file_input, file_output):
    date_pattern = re.compile(r'^\d{4}/\d{2}/\d{2} \d{2}:\d{2}:\d{2}')
    mid_line_pattern = re.compile(r'(?<!^)(\d{4}/\d{2}/\d{2} \d{2}:\d{2}:\d{2})')
    special_chars_pattern = re.compile(r'[^a-zA-Z0-9\s/:]')
    prev_line = ""

    with open(file_input, 'r') as f_in:
        with open(file_output, 'w') as f_out:
            for line in f_in:
                if not line.strip():
                    continue
                line = special_chars_pattern.sub('', line)

                if not date_pattern.match(line):
                    prev_line = prev_line.rstrip('\n') + line
                else:
                    if prev_line:
                        f_out.write(prev_line)
                    prev_line = line

                prev_line = mid_line_pattern.sub(r'\n\1', prev_line)
            if prev_line:
                f_out.write(prev_line)

file_input = "Path_OfLog"
file_output = "Path_OfPreprocessedLog"
process_file(file_input, file_output)
```

Listing 3.1: Data Cleaning Function

This data cleaning process is performed using the Python code represented in listing 3.1. This code imports the `re` library introduced in the 3.1 section and lets the developer define numerous patterns and then checks based on that. The function `data_cleaner` has two input files, the `file_input` an output file to store the new cleaned log file. The initial log file is provided as input and the output file is the clean file used for the rest of the processes.

Three defined patterns are the following:

- `date_pattern`: defines the date and timestamp pattern
- `mid_line_pattern`: defines the pattern of a line that does not start with a date
- `special_chars_pattern`: defines all the possible special characters

After initialization of the variable `prev_line` as an empty string, it is used to store the memory read-content. The input and output files are opened with the corresponding flags to provide reading from and writing to them, respectively. Then, the function iterates all lines of the input file and clears them into the output file. Suppose that the line does not start with the defined pattern of date. In that case, it is concatenated to the previous line, meaning this is the case in which a new line character is inserted before writing the memory read content. Otherwise, the line is written as is and the `prev_line` variable is updated. Then, it is checked if, inside the line, there is any date and timestamp pattern, in which case a new line is added before the date. In the end, it is written the last line is written that otherwise would never be written.

In the last lines of the algorithm, it is possible to find the variables containing the path, written as a raw string, and the call to the function. Once the memory-read log files are processed, it is possible to use them to create the dataset for the neural network.

3.5.2 Encoding

To use the memory-read log files as input for the neural network, it is necessary to encode them accordingly to extract data that is suitable for the neural network as it is needed to work with float or integer values.

Firstly, the encoding of the hash value that is needed to search was split into 64 ASCII values of the characters, and every single memory reads content into an integer since they are hexadecimal values. Then, with the sliding window technique, each sample containing the first 64 features related to the characters of the hash value is obtained, and then N features related to the memory read contents based on the dimension of the window.

The issue related to this approach is that the hexadecimal and the hash value would have a different encoding, which would lead the model to not understand the relation between the features related to the characters of the hash code and the features related to the memory read contents. This approach was pursued until the model was trained. This result is then discussed in Section 4.4.

The current model uses a different encoding to solve the first approach problem. The encoding decided to use is byte-based as each two characters represents a byte. Therefore, the hash value would be split into 32 features of two bytes each and the window would be of N-byte size, which would mean 2*N characters of the memory read log file and 1+32+N features for each sample. The first feature would contain the label related to the sample, which would be '0' for not mining activity and '1' for mining activity. Then, 32 features for the hash value and, finally, the N features of the byte-encoded memory read contents. This encoding would produce a CSV file containing small numbers that the model should be able to understand, and both the hash value and the memory read contents would have the same encoding. The model should learn the relation between the hash's features and the memory read content's features by checking its prediction using the label. The coded solution is presented in listing 3.2

```python
import csv
output_file = ’Path_Of_txt_file’
input_file = ’Path_Of_ProcessedLog’
csv_file = ’Path_Of_CSV’
first_execution = True
hash_string = ’ba184c40c8974ce60e6ec39355edaeef334b61fb614929b62e0561d789146553’
window_size=2000
step_size=100
def hex_to_int(hash_string):
    hex_pairs = [hash_string[i:i+2] for i in range(0, len(hash_string), 2)]
    decimal_values = [int(pair, 16) for pair in hex_pairs]
    return decimal_values


with open(input_file, ’r’) as infile, open(output_file, ’w’) as outfile:
    for line in infile:
        parts = line.strip().split()
        if parts:
            last_part = parts[-1]
            if ’/’ in last_part:
                last_part=last_part.replace(’/’,’’)
            if last_part and last_part[0]==’0’ and last_part!="0" and len(last_part)>1 and last_part[1]==’x’:
                last_part=last_part[2:]
                matches=[’x’,’:’,’/’]
                if not any(x in last_part for x in matches):
                    if(len(last_part)%2!=0):
                        last_part=last_part+"0"
                    outfile.write(last_part)

with open(output_file, ’r’) as infile:
        line = infile.readline()
max_start = len(line)-window_size

with open(csv_file, ’w’, newline=’’) as outfile:
        writer = csv.writer(outfile)
        field = ["mining(1=M 0=nM)"]
        for col1 in range(1, int(len(hash_string)/2)+1):
                field.append(f"hash{col1}")
        for col in range(1,int(window_size/2)+1):
            field.append(f"hex{col}")
        writer.writerow(field)

        hash_decimal_values=hex_to_int(hash_string)
        for start in range(0, max_start, step_size):
            window_stream = line[start:start + window_size]
            if "non-miners" in input_file:
                sample=[0]
            else:
                sample=[1]
            window_decimal_values=hex_to_int(window_stream)
            for value in hash_decimal_values:
                sample.append(value)
            for values in window_decimal_values:
                sample.append(values)
            writer.writerow(sample)
            sample=[]
```

Listing 3.2: Data Encoding Function

This is the algorithm used to create the CSV file with the sliding window technique and the byte encoding. At first, the variables are declared:

- `ouput_file`: contains the path for a txt file that contains all the memory read contents of the memory read log file saved in a single line of text to be encoded.

- `input_file`: contains the path of the read log file from memory that is being encoded and inserted in the CSV file.
- `csv_file`: contains the path of the CSV file that is being used to store the data set.
- `hash_string`: contains the hash value related to the memory read log file that is being encoded.
- `window_size`: contains the size of the window used to execute the sliding-window algorithm, 2000 characters in this case.
- `step_size`: contains the size of the step that is done after each window selection, 100 characters in this case.

`hex_to_int()` is a function that takes input from a string and encodes two characters at a time, one byte of information, into decimal numbers. The open function allows access to a file with read, write, or append flags. In this section of the code, every memory read content of the processed memory-read log file is extracted treating it line by line and taking that string of the line. Then some checks are done: if the string exists, if it is equal to '0', if the length is higher than 1, or if in the first and second indexes, there are '0' and 'x'. This is done to prevent any possible bug during the preprocessing execution. Then the first two characters of the memory read content are deleted, and another check is done if any special character is contained in the memory read content. After that, the read-memory content is written in the txt file. This is repeated as long as the memory-read log file is not finished.

The second open function gives access to the txt file that was just written to check the max index for the last window.

Then the CSV file is opened, for the first memory-read log file the flag is set to write but for the next executions, it is set to append, as the write function recreates the file each time it is written in. A writer variable is defined to write in the CSV file, and the field variable is used to write the features' names at first execution, after the first execution the code has to be commented to not write multiple times the features' names.

During the two cycles, the names of the columns are appended to the value variable that is then written. Then the decimal encoding of the hash value is computed, and it is needed to compute just once because it will be the same for each sample.

After starting a new cycle based on the windows, for each window, `window_size` characters are taken from the line inside the txt file. Then `sample` is declared and initialised based on whether the memory-read log file's path contains a "non-miner" directory or others. If "non-miner" is part of the path, the sample variable is appended with '0' otherwise '1'. Then the encoding of the characters that are part of the window is executed. Finally, the encoded values contained in `hash_decimal_values` and `window_decimal_values` are appended to the sample variable that is then written inside the CSV file. This process is executed repeatedly until it is impossible to define a new window of bytes. If different memory-read log files are needed inside the CSV file, the algorithm must be run as many times as the memory-read log files. As it was not developed an automated version of the code.

The dataset used for the training was created using 10 different log files, of which 3 contained mining of different types and 7 contained benign nonminers. The window used is 1000 Bytes, with a step size of 500 for mining log files and 100 for non-miner log files since the miner files are five times larger than non-miner ones. These memory read log files combined formed a 6.19GB CSV file, containing 1 845 767 samples of 1033 features each.

3.5.3 Normalization

Once the dataset is constructed, a normalization algorithm is used to normalise the data, "Normalization fosters stability in the optimization process, promoting faster convergence during gradient-based training. It mitigates issues related to vanishing or exploding gradients, allowing models to reach optimal solutions more efficiently. Normalised data are also easy to interpret and thus easier to understand. When all the features of a dataset are on the same scale, it also becomes easier to identify and visualize the relationships between different features and make meaningful comparisons" [7].

Normalization makes the training algorithm more efficient and effective. The normalization used is based on a scaler provided by the library `sklearn.preprocessing` explained in the 3.1 section.

```python
import numpy as np
import csv
import pandas as pd
from sklearn import preprocessing
csv_file='Path_Of_CSV'
hash_len_byte=32
window_size_byte=1000
standardized_csv_file='Path_Of_new_CSV'
data = pd.read_csv('csv_file', low_memory=False)
data = np.asarray(data).astype('float64')

scaler = preprocessing.StandardScaler().fit(data)
standardized_data = scaler.transorm(data)

with open('standardized_csv_file', 'w', newline='') as outfile:
    writer = csv.writer(outfile)
    field = ["mining(1=M 0=nM)"]
    for n in range(1, hash_len_byte+1):
        field.append(f"hash{n}")
    for m in range(1, window_size_byte+1):
        field.append(f"hex{m}")
    writer.writerow(field)
    writer.writerows(standardized_data)
```

Listing 3.3: Data Normalization Function

The normalisation algorithm is coded in Python and presented in listing 3.3 As explained, the data set is saved in the `data` variable and cast to a numpy array of float type. The scaler is then initialised and fitted using the data set. The scaler would learn the distribution of the dataset. Then `standardized_data` contains the dataset standardized from the scaler and is then written inside of a new CSV file together with the features' names as the first line.

The normalization produces an important problem for the byte-encoded CSV, as the CSV file already contains one byte per feature, by applying the normalization, the values become much longer, and the size of the CSV file is multiplied, leading to a file of 6 to 7 times larger. This makes the training time much longer and the model would have a very high loss during training and a 0% accuracy. Because of this situation, the model was trained on not-normalised data to check whether the calculations made with the original data could lead to a better result. The next section explains the different algorithms used in the past and how each one has different advantages and disadvantages.

3.6 Machine learning algorithm selection

As introduced in the machine learning technique section 2.1.3, different algorithms have been used to create a new model. This decision is critical for the resolution of the task, as different models have different learning mechanisms.

The decision of which algorithm to use is based on different factors: dataset and features used, time for training and inference, and the task to solve. For the solution of this thesis, as explained in Section 3.2, a classification problem is needed. Classification can be solved using supervised learning. Supervised learning is a technique in which the model makes some predictions and then checks with the real-world information provided if the prediction is right. Then, the model adjusts its weights and continues training. Using this method, the model should have more accuracy as training continues. As explained in Section 2.1.3, the most used models have been decision trees, random forests, naïve Bayes models, support vector machines, or K-nearest neighbours algorithms. For the deep learning field, it is shown how convolutional neural networks, recurrent neural networks, and long short-term networks are mostly used.

The rest of the section defines these models and the main issues related to each of them.

- **Decision tree** is a model in which the dataset is linearly divided multiple times for one feature at a time until it reaches a pure node that contains only one type of data. Decision trees are *explainable* algorithms; it is possible to understand for a human how the model decides to divide the dataset to define classes. This model is an eager learner, meaning that the model creates a predicting structure during the training, having a much longer time, and then uses this structure to output results in a very short time. The main problem related to decision trees is a high risk of *overfitting* on the training set since the predicting structure created is based only on it. Overfitting is a very well-known problem related to machine learning algorithms; it is defined as the high accuracy of the model on the training dataset and a very low accuracy on the evaluation dataset. This model may not be the correct solution for the problem as the solving algorithm must be extremely generic and not related to only the values used in the training phase.

- **Random forest** is a model created by using ensemble learning on different decision trees, meaning that different decision trees are trained and then ensembled together to create a more heterogeneous model to decrease the possibility of overfitting. This model's main problem that would lead to not being used is the computational cost and time required to train.

- **Naïve Bayes** models are a collection of classification algorithms based on *Bayes' Theorem.* The main issue related to this kind of model is the fact that it has a strong assumption on features as each pair of features classified is independent of each other, which is a very rare situation in real-world scenarios [13]. This problem makes it so it is not the correct model to use for the task.

- **Support Vector Machine (SVM)** is a *supervised machine learning algorithm* based on the division of a dataset for classification or regression tasks. It is based on finding a line or hyper-plane to divide the data points in the dataset to classify them [14]. This model's main issue is related to the fact that it is not suited for large datasets as it is in this thesis' situation [15].

- **The k-nearest neighbors (KNN)** is an algorithm that is based on classifying different data points based on its k nearest data points' class. It is one of the easiest algorithms to implement. It is based on computing the distance of a new data point to the others already classified, then based on which are the closest ones, it is given a class [16], this algorithm's main issue is related to the fact that it is a lazy learner, it means that each time the analysis of a new sample is requested it needs to run back the algorithm on the entire dataset, which would be a problem as the dataset would contain big amounts of data, making it impossible to use for this thesis' solution.

- **Convolutional neural networks** are a kind of neural network based on layers that have different jobs inside of the neural network, such as convolutional, pooling, and fully connected.

Convolutional neural networks are mostly used for image detection as they work with a kernel matrix to check for usual or unusual features inside images. This kind of neural network is used mainly in computer vision when the tasks require solving a problem related to visual data or images. This could be a possible solution for future implementations but requires a more complex analysis.

- **Recurrent neural networks** are neural networks based on recursion to keep a trace of memory for the latest samples analyzed, this makes it so the models learn to classify through information stored from previous samples. This kind of neural network works better with sequential data.

After analyzing the different described algorithms, it is shown how models to perform well need a well-defined problem to solve, which is itself a very difficult task, as a matter of fact, choosing the right model is extremely complicated. For this thesis work, it was chosen to use a neural network. This choice was based on the fact that this algorithm would be the best starting idea to develop the first results, as it is easy to create and would work better with the new data set, as memory-read log files can be extremely difficult to elaborate.

4 Experimental evaluation

This chapter describes the training and testing algorithms and focuses on the analysis of the results obtained.

4.1 Deployment environment

The machine provided by the university for the development and training of the machine learning model has the following specifications:

- Operating System: Ubuntu 22.04.4 LTS
- Graphics Processor: NVIDIA RTX A5000
- CUDA Cores: 8192
- Total Dedicated Memory: 24564MB
- Memory Interface: 384-bit
- PCIe Generation: Gen3
- Maximum PCIe Link Width: x16

The machine is utilized for storing and executing programs, as machine learning algorithms often require significant computational power. The large RAM capacity is essential for efficiently handling large datasets. All the libraries mentioned in the previous section are installed on the machine, and the memory-read log files are imported for further processing.

4.2 Evaluation dataset

This thesis is based on CryptojackingTrap's influence in the cryptojacking detection field. The starting point for creating the model would be CryptojackingTrap's dataset. CryptojackingTrap's dataset [33], which was used to evaluate the algorithm in the original work, comprises different memory-read log files and the corresponding hash values for each. Unfortunately, the dataset does not contain actual mining malware as in the previous study CryptojackingTrap it is mentioned that the known sites that would run malicious scripts have ceased cryptojacking activities. An example is Coinhive which after its closure in March 2019 could not be used anymore for cryptojacking analysis. Even so, our goal is to detect if mining is happening, without assuming what kind of mining it is, benign miners' data is enough for training the model.

The memory-read log files are divided into three categories:

- **benign miners:** Containing the memory-read logs of different mining applications for different cryptocurrencies, Bitcoin - CpuMiner, Monero - MinerGate, Monero - XMRig.
- **benign non-miners:** Containing common application's memory-read logs, Adobe Acrobat DC, Microsoft Calculator, Microsoft Word, Notepad++, Photos.
- **generated dataset:** This category contains randomized memory-read logs, expanded benign miner read logs, and randomized benign miner.

For this thesis task, the focus was mainly on the benign miners and benign non-miners to train the model. The benign miners' dataset contains 8 memory-read log files, each containing around 3 to 30 million lines of memory-read content retrieved from the memory by the processes. The benign non-miners category contains 30 different log files, 6 for each application monitored, these would be smaller in size compared to the mining ones.

4.3 Training and testing

This section explains the supervised training algorithm and the testing algorithm used on different datasets. The algorithm used for creating, training, validating, and saving the model is presented in listing 4.1.

```python
import pandas as pd
import numpy as np
import keras
from sklearn.model_selection import train_test_split
import tensorflow as tf
csv_file='Path_Of_CSV'
dataset = pd.read_csv(csv_file, low_memory=False)
x = dataset.drop(columns=["mining(1=M 0=nM)"])
y = dataset["mining(1=M 0=nM)"]
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.5)
x_train = np.asarray(x_train).astype('float64')
x_test = np.asarray(x_test).astype('float64')

model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(1033, activation='sigmoid'))
model.add(tf.keras.layers.Dense(128, activation='sigmoid'))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
model.compile(optimizer='adam', loss='BinaryCrossentropy', metrics=['accuracy'])

model.fit(x_train, y_train, epochs=20)
model.evaluate(x_test, y_test)
model.save('AICryptojackingTrapModel.keras')
```

Listing 4.1: Training Algorithm

`standardized_csv_file` contains the dataset's path, it is so read using pandas and saved inside of `dataset` variable. After that, the dataset is split into `y` which contains the column of the labels, and `x` which contains all the other columns. After that, both are split:

- `x_train`: contains the columns used for the training.
- `x_test`: contains the columns used for the evaluation.
- `y_train`: contains the relative labels of the `x_train` columns
- `y_test`: contains the relative labels of the `x_test` columns

After that both the training and test are cast into numpy arrays of float type. Then the neural network is defined, using a Sequential model and three dense layers, the input layer made of 1033 neurons, one for each feature, a hidden layer containing 128 neurons, and the output layer containing just one neuron to represent the probability of the output '1' or '0'. The model is then compiled using `adam` optimizer: "Algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning" [21]. `BinaryCrossentropy` as a loss function, that is used to make binary predictions, and outputting the `accuracy` as a metric. The model is then trained using the training dataset for 20 epochs. Then, it is validated on the validation dataset and saved in the directory. Then the testing is done on two new memory-read log files to check the model's results on completely new data. In listing 4.2 the program executed for testing.

```python
import keras
import tensorflow as tf
import numpy as np
import pandas as pd
path_model='Path_Of_Model'
path_testing='Path_Of_testing_CSV'
path_testing1='Path_Of_secondTesting_CSV'
model = tf.keras.models.load_model(path_model)
dataset = pd.read_csv(path_testing)
x = dataset.drop(columns=["mining(1=M 0=nM)"])
y = dataset["mining(1=M 0=nM)"]
model.evaluate(x,y)

dataset1 = pd.read_csv(path_testing1)
x1 = dataset1.drop(columns=["mining(1=M 0=nM)"])
y1 = dataset1["mining(1=M 0=nM)"]
model.evaluate(x1,y1)
```

Listing 4.2: Testing Algorithm

This program is executed to check the predictions made by the model trained on a mining CSV file and a not mining CSV file, both are created using the algorithm explained in Section 3.5.2.

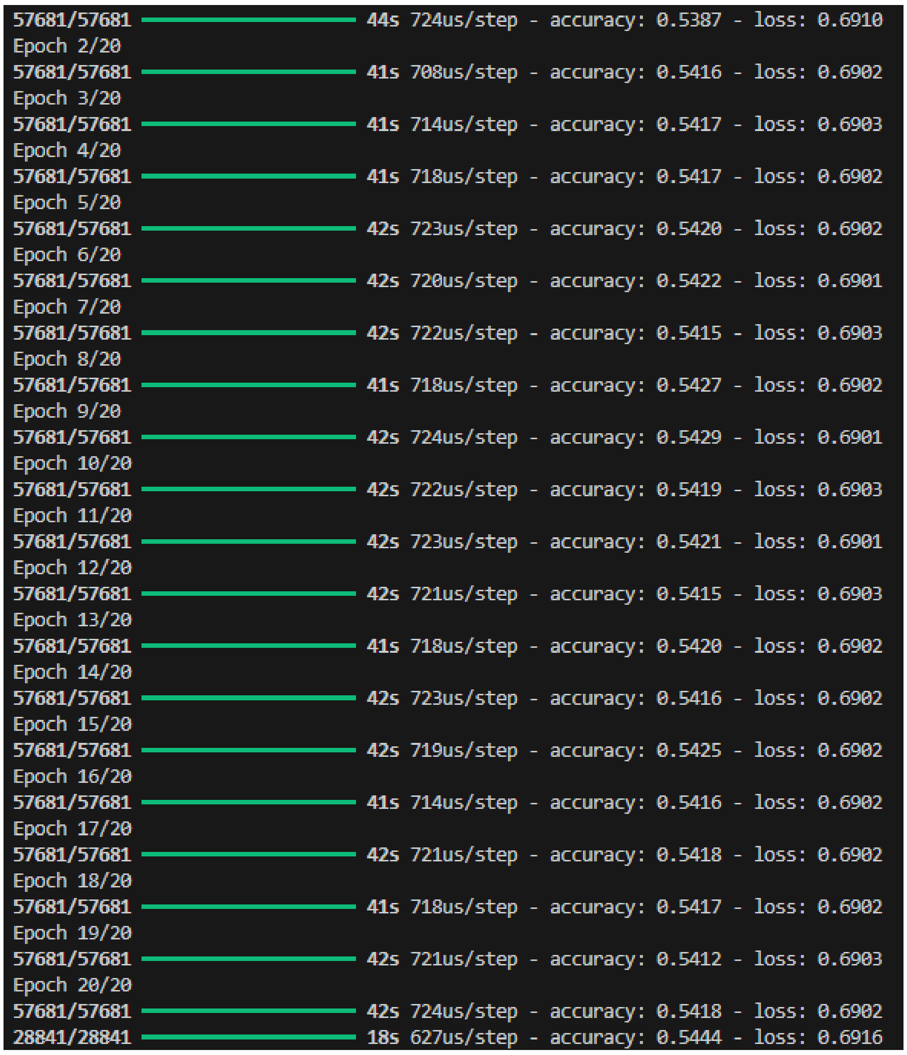
First are defined the variables `path_model`, `path_testing`, and `path_testing1`, which contain the model's path, the mining testing dataset's path, and the non-mining testing dataset's path. Then, it is imported the model saved before using the `model.save()` function, by calling the `models.load_model()` function. After that, the first dataset is split based on the label's column name, and the model is then tested on the first dataset. The same is done for the second dataset. The next section shows and explains the results that are obtained using these algorithms.

4.4 Results

This section shows the results obtained after following the ideas explained in the previous chapters, and how they were used to understand the problems related to the approaches pursued.

As explained in Section 3.5.2, the first approach, in which the memory read contents and the hash value had different encoding, was pursued until training and gave a result of a model having 35% accuracy in the training phase, this result led to a deeper analysis of what was happening inside of the model. The issue found in this approach was that the memory read contents, encoded into integers, would be extremely large numbers that had to be treated. After the normalization of the data, the model would not have any improvement. This approach showed how it was not possible to interact with such different features. It was based on this result that the encoding was changed, as it was understood that the memory read contents could not be treated as integers while the hash value would be represented with very small ones.

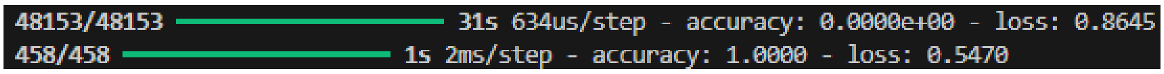
The model trained with the byte encoding dataset has an accuracy of 54% and a loss of 69% during the training phase. These improved results show that the model still does not understand the pattern between the hash features and the hexadecimal features. In Fig. 4.1 are shown the epochs of the training phase. It is shown how the loss is not reducing the more the model is trained, exposing that there is no learning in the process. The last line of output is the validation done, using a portion of the starting dataset that was not used for the training. The accuracy of 54% and the loss of 69% given from this evaluation is based on the fact that the model is learning the training set, but it needs further testing on completely different data to be sure the model is learning a pattern that can be used in the real world. This is required because even if the portion of the CSV file used for the evaluation is not used for the training, it still contains similar data to the one used during the training phase.



```
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 44s 724us/step - accuracy: 0.5387 - loss: 0.6910
Epoch 2/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 41s 708us/step - accuracy: 0.5416 - loss: 0.6902
Epoch 3/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 41s 714us/step - accuracy: 0.5417 - loss: 0.6903
Epoch 4/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 41s 718us/step - accuracy: 0.5417 - loss: 0.6902
Epoch 5/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 723us/step - accuracy: 0.5420 - loss: 0.6902
Epoch 6/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 720us/step - accuracy: 0.5422 - loss: 0.6901
Epoch 7/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 722us/step - accuracy: 0.5415 - loss: 0.6903
Epoch 8/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 41s 718us/step - accuracy: 0.5427 - loss: 0.6902
Epoch 9/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 724us/step - accuracy: 0.5429 - loss: 0.6901
Epoch 10/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 722us/step - accuracy: 0.5419 - loss: 0.6903
Epoch 11/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 723us/step - accuracy: 0.5421 - loss: 0.6901
Epoch 12/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 721us/step - accuracy: 0.5415 - loss: 0.6903
Epoch 13/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 41s 718us/step - accuracy: 0.5420 - loss: 0.6902
Epoch 14/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 723us/step - accuracy: 0.5416 - loss: 0.6902
Epoch 15/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 719us/step - accuracy: 0.5425 - loss: 0.6902
Epoch 16/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 41s 714us/step - accuracy: 0.5416 - loss: 0.6902
Epoch 17/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 721us/step - accuracy: 0.5418 - loss: 0.6902
Epoch 18/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 41s 718us/step - accuracy: 0.5417 - loss: 0.6902
Epoch 19/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 721us/step - accuracy: 0.5412 - loss: 0.6903
Epoch 20/20
57681/57681 ━━━━━━━━━━━━━━━━━━━━ 42s 724us/step - accuracy: 0.5418 - loss: 0.6902
28841/28841 ━━━━━━━━━━━━━━━━━━━━ 18s 627us/step - accuracy: 0.5444 - loss: 0.6916
```

Figure 4.1: Output of the training program

Then, in Fig. 4.2 is shown the evaluation done with the testing program, the model has a 100% accuracy and 55% loss on not-mining memory-read log files and 0% accuracy and 87% loss on mining memory-read log files, showing that the model is predicting as not-mining every time, showing that the previous analysis is correct and the learning has not been effective.



```
48153/48153 ━━━━━━━━━━━━━━━━━━━━ 31s 634us/step - accuracy: 0.0000e+00 - loss: 0.8645
458/458 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 1.0000 - loss: 0.5470
```

Figure 4.2: Output of the testing program

The results show that the model has not solved the task, as the memory-read log files still contain too much irrelevant information for the model to understand what is relevant. In the next chapter, some ideas are suggested to explore a more complex solution that could solve this problem and create a model that understands what these memory-read log files can offer for faster detection.

5 Conclusions and future work

This chapter concludes the work done and explains the possible future steps. This thesis aims to start a new development of the CrytpojakingTrap work, as the novel idea used to detect cryptojacking is crucial as it is resilient to most techniques that cyber-criminals use to evade security checks. The thesis explains how different ideas were pursued to reduce the original architecture's computational time by implementing a machine-learning model. It is shown how the problem has not been entirely solved, and it requires a deeper knowledge of memory-read log files to understand how useful they can be. The model produced has an accuracy of 54% and 69% loss during the training, while during the testing phase, 100% accuracy on non-mining data and 0% accuracy on mining data, showing that the model did not understand the pattern that defines cryptojacking inside the dataset. The code to produce the dataset, train, and test the model was implemented from scratch and explained. This work is just the start of the much more complex and important studies possible to make on the architecture that is wanted to improve. This thesis is the first research done on this kind of resilient detection, based on machine learning and memory reads, and this work overcomes different challenges unique to the problem that is set to be solved. The problems that were described in this thesis should be important information for the next steps, as most of these are intrinsic to the memory-read log files. In the following sections are given some possible ideas that could be used to develop an improved solution to solve the task.

5.1 Future work

The results reached, as much as they are not final, show that there is potential for further improvement, and it should be possible to obtain better results with a more complex structure, such as deep neural networks or different machine learning models. These structures could be able to define what is relevant information inside of the memory-read log files and make the original work algorithm more efficient. Different models could be used to understand what information is needed to read from the memory-read log files, or another approach could be the following: the model could be used to improve a single case in which the original architecture has a very expensive computational cost. Therefore, the model could be used only in specific cases during the execution. This would make it so the training of the model could be relative to only a specific case of memory-read log files, reducing the amount of information needed for the training and the possible related problems. Then, different models could be used to check which one is detecting mining.

A possible aspiration for future works could be relative to new algorithms for the models and new algorithms for encoding the memory-read log files. The algorithms explained in this thesis could be an initial point to start a deeper analysis that could be based even on the timestamps that were not used for this solution. As it is explained, with the per-byte encoding, the normalization would make the file critically larger and make the learning impossible as the model would not understand. It could be possible to create more precise CSV files by changing the encoding, which, with a smaller amount of data, can keep the same amount of information. A possible idea could be to use a Principal Component Analysis algorithm to reduce the number of dimensions of the model and maintain the information. These ideas are just some of the possibilities that could be used to develop the work, as machine learning is vast and full of different possibilities.

Bibliography

[1] Google Brain. tf.keras.layers.dense. https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense, 2024.

[2] Google Brain. tf.keras.model. https://www.tensorflow.org/api_docs/python/tf/keras/Model, 2024.

[3] Google Brain. tf.keras.sequential. https://www.tensorflow.org/api_docs/python/tf/keras/Sequential, 2024.

[4] Atefeh Zareh Chahoki, Hamid Reza Shahriari, and Marco Roveri. Cryptojackingtrap: An evasion resilient nature-inspired algorithm to detect cryptojacking malware. *IEEE Transactions on Information Forensics and Security*, pages 1–1, 2024.

[5] Contabo. What is cryptojacking. https://contabo.com/blog/what-is-cryptojacking, 2024.

[6] Coursera. What is tensorflow? definition, use cases, and more. https://www.coursera.org/articles/what-is-tensorflow, 2024.

[7] datacamp. Why normalize data. https://www.datacamp.com/tutorial/normalization-in-machine-learning, 2024.

[8] NumPy Developers. Numpy documentation. https://numpy.org/doc/stable, 2024.

[9] docs.python.org. csv — csv file reading and writing. https://docs.python.org/3/library/csv.html, 2024.

[10] docs.python.org. re — regular expression operations. https://docs.python.org/3/library/re.html, 2024.

[11] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. A first look at browser-based cryptojacking. 03 2018.

[12] GeekForGeeks. Sliding window technique. https://www.geeksforgeeks.org/window-sliding-technique/, 2024.

[13] GeeksForGeeks. Naive bayes classifiers. https://www.geeksforgeeks.org/naive-bayes-classifiers/, 2024.

[14] GeeksForGeeks. Support vector machine. https://www.geeksforgeeks.org/support-vector-machine-algorithm/, 2024.

[15] GeeksForGeeks. Support vector machine. https://www.geeksforgeeks.org/support-vector-machine-in-machine-learning/, 2024.

[16] IBM. What is the knn algorithm? https://www.ibm.com/topics/knn, 2024.

[17] Intel. Pin - a dynamic binary instrumentation tool. https://www.intel.com/content/www/us/en/developer/articl a-dynamic-binary-instrumentation tool.html, 2024.

[18] Laith M Kadhum, Ahmad Firdaus, Syifak Izhar Hisham, Waheed Mushtaq, and Mohd Faizal Ab Razak. Features, analysis techniques, and detection methods of cryptojacking malware: A survey. *JOIV: International Journal on Informatics Visualization*, 8(2):891–896, 2024.

[19] Keras. About keras 3. https://keras.io/about, 2024.

[20] Sheharbano Khattak, Naurin Rasheed Ramay, Kamran Riaz Khan, Affan A. Syed, and Syed Ali Khayam. A taxonomy of botnet behavior, detection, and defense. *IEEE Communications Surveys & Tutorials*, 16(2):898–924, 2014.

[21] Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[22] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430, 2007.

[23] Otavio Kiyatake Nicesio and Adriano Galindo Leal. A systematic literature review of machine learning approaches for in-browser cryptojacking detection. In *2023 7th Cyber Security in Networking Conference (CSNet)*, pages 102–108. IEEE, 2023.

[24] scikit-learn developers. Getting started. https://scikit-learn.org/stable/getting_started.html, 2024.

[25] SonicWall. 2023 sonicwall cyber threat report. Technical report, SonicWall, 2023.

[26] SonicWall. 2024 sonicwall mid-year cyber threat report. Technical report, SonicWall, 2024.

[27] TensorFlow. tf.keras.models.load_model. https://www.tensorflow.org/api_docs/python/tf/keras/models/lo
2024.

[28] TensorFlow.org. Save and load models. https://www.tensorflow.org/tutorials/keras/save_and_load, 2024.

[29] TwinGate. What is a bot master. https://www.twingate.com/blog/glossary/bot-master, 2024.

[30] W3.school. Pandas introduction. https://www.w3schools.com/python/pandas/pandas_intro.asp, 2024.

[31] Guangquan Xu, Wenyu Dong, Jun Xing, Wenqing Lei, Jian Liu, Lixiao Gong, Meiqi Feng, Xi Zheng, and Shaoying Liu. Delay-cj: A novel cryptojacking covert attack method based on delayed strategy and its detection. *Digital Communications and Networks*, 9(5):1169–1179, 2023.

[32] Atefeh Zareh and Hamid Reza Shahriari. Botcointrap: Detection of bitcoin miner botnet using host based approach. In *2018 15th International ISC (Iranian Society of Cryptology) Conference on Information Security and Cryptology (ISCISC)*, pages 1–6, 2018.

[33] Atefeh Zareh Chahoki. Cryptojackingtrap dataset. https://ieee-dataport.org/documents/cryptojackingtrap, 2023.