POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea

Automatic Malware Signature Generation

Relatori

prof. Antonio Lioy

ing. Andrea Atzeni

Michele Crepaldi

Anno accademico 2020-2021

Thanks...

Summary

In most recent years the proliferation of malicious software, namely Malware, has had a massive increase: according to AV Atlas Dashboard [1] the new malware samples (and PUA - Potentially Unwanted Application) currently detected every day are about 440.000 (at the time of writing), and this number is predicted to only keep growing. The total number of known Microsoft Windows malicious software (and PUA) passed from about 55 million in 2011, to about 400 million in 2016, and finally to nearly 830 million now. The huge number of malware samples out there in the wild renders the detection through manually generated signatures (patterns that identify malicious code) infeasible and consequently imposes the urgent need of tools able to automatically detect malware and possibly describe it in an human-interpretable way. Several methodologies have been proposed through the years, ranging from signature-based detection (especially with Yara Rules) to various Machine Learning approaches like Decision Trees, Naive Bayes models and Neural Networks.

This thesis presents a novel model built upon previous works in **ML-based** (Machine Learning) automatic PE (Microsoft Windows Portable Executable) malware detection and description and introduces a new evaluation procedure on the learned implicit representation/signature of malware samples that may prove the applicability of its usage in the Malicious family prediction and ranking tasks. The model is trained on an open source large scale dataset of malware and benignware samples with the aim of creating high quality implicit signatures capable of correctly detecting (and describing) unseen malware samples as well as obfuscated malware and new variants, with high True Positive Rate (TPR) and high Recall at low False Positive Rates (FPRs).

The **Proposed Model**'s results in the different tasks (both the ones it was trained on - Malicious/Benign label and descriptive tags prediction - and the additional malware family prediction and ranking tasks) were compared to the previous models' ones. In particular, the **ALOHA** model proposed in [2] and the **Joint Embedding** model described in [3] were selected as reference models.

The results show that the **Proposed Model** generates implicit signatures (samples embeddings) which provide comparable TPRs, Accuracies, Recalls, Precisions and F1 Scores at low false positive rates with respect to the ones produced by the mentioned previous methods on the corresponding tasks. When testing the **Proposed Model**'s learned representations on the Malware Family prediction and ranking tasks, however, the results were less promising. Therefore, a new Malware **Family Classifier** model was created and trained (and evaluated) on a specially crafted dataset of 10.000 PE files, exploiting the parameters from a previous **Proposed Model** training run, with the aid of transfer learning.

Future (and more performant) works capable of overcoming some of the resulting model limitations may be very useful to the IT-Security field in the current scenario and could even enable the generation of explicit (and thus more interpretable) signatures derived from the learned implicit ones.

Acknowledgements

Aknowledgments...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Malware first made its appearance in the 1960s. Then, hackers used to design computer viruses mainly for fun, as an exciting prank/experiment; their creations would generally display harmless messages and then spread to other computers [4]. There are numerous examples of malware created at that time within a laboratory setting: for example the ***Darwin game*** in 1962 and ***Creeper*** in 1971.

In the early 1980s, the concept of malware caught on in the technology industry, and numerous examples of viruses and worms appeared both on Apple and IBM personal computers. With the introduction of the World Wide Web and the commercial internet in the 1990s it eventually became widely popularized, so much that Yisreal Radai coined the term **malware** in 1990.

```
Elk Cloner:  The program with a personality

   It will get on all your disks
     It will infiltrate your chips
       Yes it's Cloner!

   It will stick to you like glue
     It will modify ram too
       Send in the Cloner!
```

Figure 1.1: Poem displayed by Elk Cloner virus

The previously mentioned 1960s and 1970s malware were all kept within a laboratory environment and never managed to escape to the wild. ***Elk Cloner*** (1981) was the first known virus to have been able to escape its creation environment. It spread through infected floppy disks but did not cause deliberate harm simply displaying the 'poem' shown in figure 1.1 on the user's screen. Following the success of that prank gone wild, the first Microsoft PC virus, called ***Brain***, was then created in 1986. Again, like *Elk Cloner*, Brain was mostly annoying rather than harmful, but it was also the first known virus capable of concealing its presence on the disk thus evading detection. In 1988 the first worm, called ***Morris*** worm, an experimental, self-propagating, self-replicating program was released on the internet [5]. In 1988 made its appearance also the first example of intentionally harmful virus, the ***Vienna*** virus, which encrypted data and destroyed files. This led to the creation of the first antivirus tool ever [4].

In the following decades malware has evolved both regarding its complexity and malware sample numbers. In particular, according to AV Atlas [1], in 2019 and 2020 (and until mid 2021 - that is until the time of writing) the number of newly generated malware blew with respect to previous years, as shown in graph 1.2, to the point that currently approximately 5.1 new Microsoft Windows malware (and PUA) are generated per second, $\sim$ 18,500 per hour. As reported by AV Atlas dashboard and as shown in graph 1.3, in recent years the total amount of unique malware (and PUA) variants reached impressive numbers, to the point that they are now more than

830 millions. Moreover, nowadays malware commonly use obfuscation and other sophisticated techniques, such as Polymorphism and Metamorphism, to evolve their structure thus evading detection.

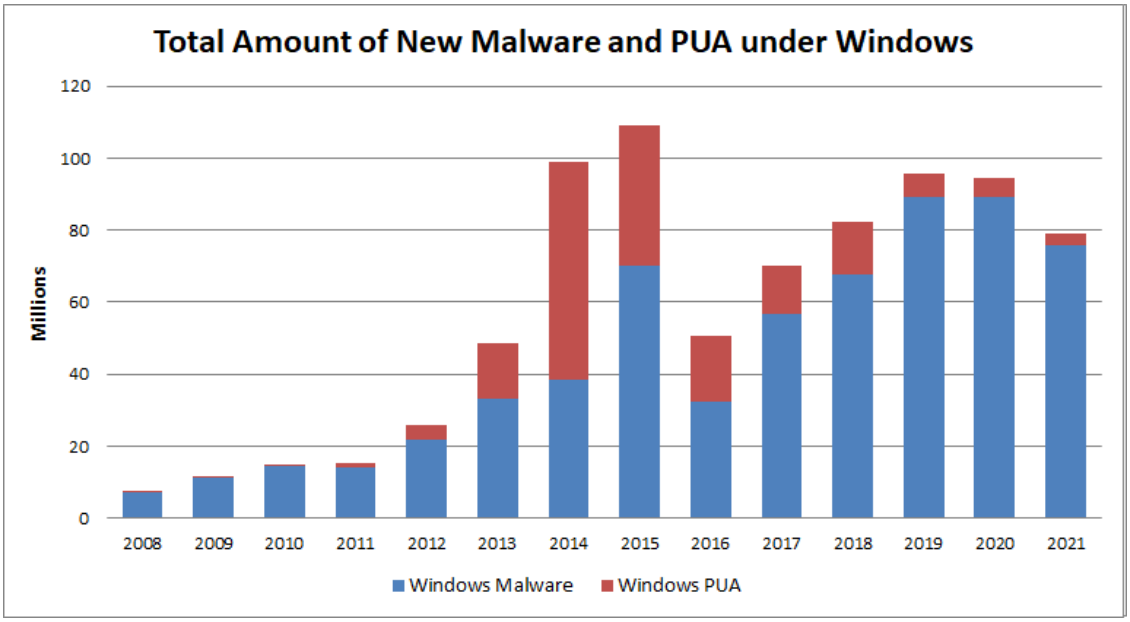


Figure 1.2: AV Atlas - Total Amount of New Malware and PUA

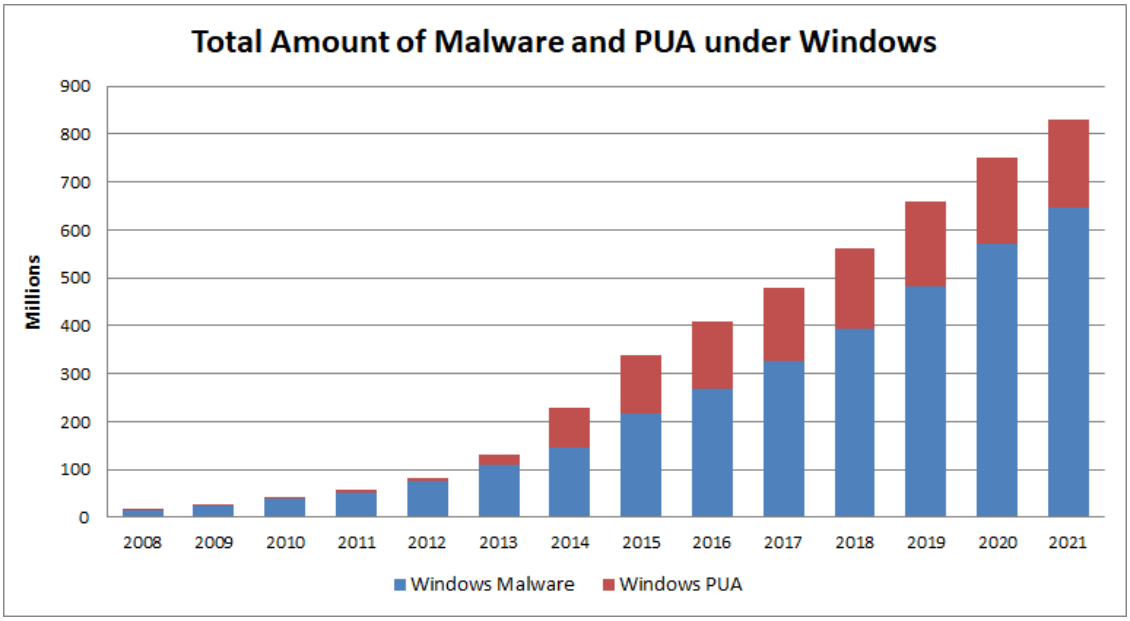


Figure 1.3: AV Atlas - Total Amount of Malware and PUA

For all these reasons, signature-based detection techniques, such as manually generated Yara Rules, which are typically used by most commercial anti-virus solutions, are becoming inefficient in the present scenario. In fact, it is now straight up impossible for analysts to manually analyse each malware variant that is found in the wild. Moreover, even when a new malware family is identified and an appropriate amount of its samples are analysed, the generated signature may not be capable of detecting new variations or may even be rendered useless through the use of obfuscation and/or polymorphic mechanisms. There is therefore the need for automated malware analysis solutions capable of automatically generating (implicit or explicit) signatures effective at

distinguishing malicious from benign code while being less susceptible to code modifications and obfuscation attempts.

This thesis presents a research aimed at satisfying this need for automated malware detection solutions. In particular, it presents a novel model built upon previous works on ML-based (Machine Learning) automatic malware detection and description designed for PE (Microsoft Windows Portable Executable) files. Moreover, it introduces a new evaluation procedure that may prove the applicability of the model learned implicit representation/signature of malware samples in the Malware family prediction and ranking tasks. These tasks are particularly interesting for malware analysts since they allow them to quickly categorize malware samples as being part of specific sets (families) of samples with common behavioural and structural characteristics.

The proposed framework life cycle can be conceptually divided in four phases (as shown in figure 1.4): **model architecture definition**, **model training and validation**, **model evaluation** and finally **model deployment**. In particular, in the first phase the proposed FNN (Feedforward Neural Network) model architecture is defined and implemented taking inspiration from previous works such as the **ALOHA** and the **Joint Embedding** models presented in papers [2] and [3], respectively. In the second phase, instead, the proposed model is trained (and validated) on an open source large scale dataset of malware and benignware samples (Sorel20M [6]) with the aim of creating high quality implicit signatures capable of detecting (and describing via SMART tags) unseen malware samples, as well as obfuscated malware and new variants, with high True Positive Rate (TPR) and high Recall at low False Positive Rates (FPRs). The first two phases here described are iteratively repeated until a model with satisfactory training and validation loss trends is generated. In the third phase, on the other hand, the final model architecture is tested on the Malware detection and Description tasks and the corresponding prediction scores are computed and plotted. Moreover, in this phase the model learned representation of PE files is also tested on the Malware family prediction and ranking tasks using a novel dataset, referred to as '*fresh dataset*' in this document, containing 10.000 samples belonging to 10 of the most widespread malware families in Italy at the time of writing, specifically created for that purpose. In both datasets the samples are directly represented by their numerical feature representation extracted statically from specific fields of each sample's Windows Portable Executable (PE) file header. The proposed model thus relies exclusively on static analysis features which are generally simpler, less computationally intensive and thus faster than dynamic analysis ones (executable's behaviour characteristics). Finally, in the last phase the final proposed model architecture is deployed in the wild. In particular, it can be used as an automatic Malware detection tool that provides additional description tags useful for remediation. Moreover, potentially, if the corresponding evaluation results allow it, it could also be used to provide information about the malware family each analysed sample most probably belongs to, among the set of families of interest.

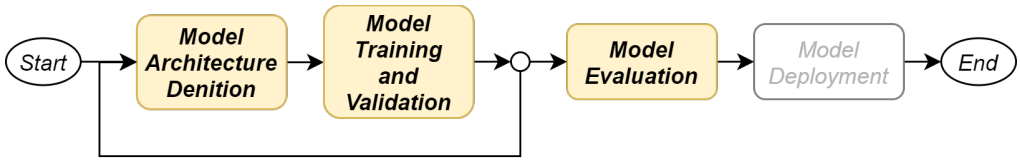


Figure 1.4: Proposed Framework Life Cycle

This thesis focuses on the first three phases previously mentioned. In particular, it concentrates on defining, training and evaluating the best model architecture possible for the tasks at hand. However, some code optimization challenges resulting from the slowness of the code in the instance used for the experiments meant that it could be possible to train the model only with the first half of the samples of the Sorel20M dataset in a reasonable time, with some approximations on the samples dispersion when random sampling them from the dataset. This led to slightly worse performance than what is theoretically possible with the current architecture using the whole dataset. Nevertheless, the deployment of the proposed model on a real-world scenario is theoretically possible with the current final architecture, although it would be better to train the proposed model on the whole Sorel20M dataset on a better instance first in order to see its true potential.

At a later moment, the proposed framework was extended with the addition of a **Malware Family Classifier** model head defined on top of the **Proposed Model** base topology in order to improve its relatively poor results in the Malware Family Prediction task. This new model was then specifically trained (and tested) for such purpose on the relatively small '*fresh dataset*', which contains information about the malware family each sample belongs to. However, instead of training the newly defined architecture from scratch on such small dataset at the risk of overfitting, the technique called '*Transfer Learning*' was used by transferring the knowledge (the learned model parameters) from a previous **Proposed Model** training run on the large Sorel20M dataset onto the new model base topology (the one shared with the **Proposed Model** architecture), before training. Then, during the training procedure some of the imported parameters were 'fine-tuned' while the ones corresponding to the newly added Family Classifier head were learned from scratch.

Chapter 2

Background

2.1 Malware

Malware, short for ***malicious software***, is a general term for all types of programs designed to perform harmful or undesirable actions on a system. In fact in the context of IT security the term *malicious software* commonly means [7]:

> *Software which is used with the aim of attempting to breach a computer system's security policy with respect to Confidentiality, Integrity and/or Availability.*

Malware consists of programming artefacts (code, scripts, active content, and other software) designed to disrupt or deny operation, gain unauthorized access to system resources, gather information that leads to loss of privacy or exploitation, and other abusive behaviour. Malware is not (and should not be confused with) defective software - software that has a legitimate purpose but contains harmful bugs (programming errors).

Different companies, organizations and people describe malware in various ways. For example **Microsoft** defines it in a generic way:

> *Malware is a catch-all term to refer to any software designed to cause damage to a single computer, server, or computer network* [8].

The **National Institute of Standards and Technology** (**NIST**), on the other hand, presents multiple definitions for malware, describing it as "hardware, firmware, or software that is intentionally included or inserted in a system for a harmful purpose" [9].

In another more specific definition **NIST** affirms that Malware is:

> *A program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or operating system or of otherwise annoying or disrupting the victim* [9].

The computer system whose security policy is attempted to be breached is usually known as the ***target*** for the malware. The cybercriminal who originally launched the malware with the purpose of attacking one or more targets, on the other hand, is generally referred to as the "*initiator* of the malware". Furthermore, depending on the malware type, the initiator may or may not exactly know what the set of targets is [7].

According to the above definitions software is defined as malicious in relation to an attempted breach of the target's ***security policy***. In other words, software is often identified as malware based on its *intended use*, rather than the particular technique or technology used to build it.

2.1.1 Why is Malware used

Generally, cybercriminals use malware to access targets' sensitive data, extort ransoms, or simply cause as much damage as possible to the affected systems.

More generally malware serves a variety of purposes. For example, the most common cybercriminals' uses of malware are: [10]

- **To profit financially (either directly or through the sale of their products or services)**. For example, attackers may use malware to infect targets' devices with the purpose of stealing their credit account information or cryptocurrency. Alternatively, they may sell their malware to other cybercriminals or as a service offering (*malware-as-a-service*).
- **As a means of revenge or to carry out a personal agenda**. For example, Brian Krebs of Krebs on Security was struck by a big DDoS attack in 2016 after having talked about a DDoS attacker on his blog.
- **To carry out a political or social agenda**. For example, there exist many Nation-state actors (such as state-run hacker groups in China and North Korea) and hacker groups such as Anonymous.
- **As a way to entertain themselves**. Some cybercriminals perpetrate attacks on victims just for fun.

Obviously there are also reasons for **non**-malicious actors to create and/or deploy some types of malware too - for example they can be used to test a system's security, to spy on someone as part of a legal act/police operation, etc.

2.1.2 Common Malware types

There are different ways of categorizing malware; one way is by *how* the malicious software spreads. Another one is by *what* it *does* once it has successfully infected its victim's computers (i.e. what is its payload, how it exploits the system or makes it vulnerable).

By how they spread

Terms like *trojan*, *virus* and *worm* are commonly used interchangeably to indicate generic malware, but they actually describe three subtly different ways malicious software can infect target computers [11]:

- ***Trojan horse***. Generally speaking, a *Trojan Horse*, commonly referred to as a "Trojan", is any program that disguises itself as legitimate and invites the user to download and run it, concealing a malicious payload. When executed, the payload - malicious routines - may immediately take effect and cause many undesirable effects, such as deleting the user files or installing additional malware or PUAs (Potentially Unwanted Apps), depending on the payload attached to them [5].

  Trojans may hide in games, apps, or even software patches, or they may rely on social engineering and be embedded in attachments included in phishing emails. However, they cannot self-replicate but rely on the system operators to activate.

- ***Virus***. The term "computer virus" is used for describing a **passive** self-replicating malicious program. Usually spread via infected websites, file sharing, or email attachment downloads, it lies dormant until the infected host file or program is activated. At that point it spreads to other executables (and/or boot sectors) by embedding copies of itself into those files. A virus, in fact, in order to spread from one computer to another, usually relies on the infected files possibly ending up, by some means or another, in the target system. The mean of transport (file, media file, network file, etc.) is often referred to as the virus

vector. Depending on how complex the virus code is, it may be able to modify its copies upon replication. For the transport of the infected files to the target system(s), the virus may rely on an unsuspecting human user (who for example uses a USB drive containing the infected file) or may initiate the transfer itself (for example, it may send the infected files as an e-mail attachment) [7].

Viruses may also perform other harmful actions other than just replicating, such as creating a backdoor for later use, damaging files, stealing information, creating botnets, render advertisements or even damaging equipment.

- ***Worm***. A worm is a self-replicating, **active** malicious program that exploits various system vulnerabilities to spread over the network. Particularly, it relies on vulnerabilities present in the target's operating system or installed software. Worms usually consume a lot of bandwidth and processing resources due to continuous scanning and may render the host unstable, sometimes even causing crashes. Apart from replicating, computer worms also contain "payloads" which are pieces of code written to perform various nefarious actions on the affected hosts, for example stealing data, deleting files or creating bots - which can lead the infected systems to become part of a botnet [5].

Moreover, attackers can also install malware "manually" on a computer, either by gaining physical access to the target system or by using privilege escalation methods to obtain remote administrator access [12].

By what they do

There is a wide range of potential attack techniques used by malware, here are briefly described some of them:

- ***Adware***. *Adware* is any software package which automatically plays, displays, or downloads advertisements to a computer. These can be in the form of pop-up ads, ad banners in websites, or advertisements displayed by software, that lure the user into making a purchase. The goal of Adware is, in fact, to generate revenue for its author.

  Often times adware comes even bundled with "free", or discounted, versions of non-malicious software and/or applications since it is usually seen by developers as a way to recover development costs [13].

  Adware, by itself, is annoying but somewhat harmless, since it is solely designed to deliver ads; however, it often comes bundled with spyware (such as keyloggers), and/or other privacy-invasive software that is capable of tracking user activity and steal information [14].

- ***Backdoor***. A *backdoor*, also called Remote Access Trojan (RAT), is a vulnerability deliberately buried into software's code allowing to bypass typical protection mechanisms, like credentials-based login authentication. Once a system has been compromised (by others types of malware or other methods), one or more backdoors may be installed. This is done with the purpose of allowing the attacker easier access in the future without alerting the user or the system's security programs. Moreover, backdoors may also be installed before other malicious software, to allow attackers entry [13].

  On the other hand, it is not uncommon to see many (non-malicious) device or software manufacturers ship their products with intentionally hidden backdoors to allow company personnel or law enforcement access to the system when needed [15]. Alternatively, backdoors are sometimes hidden in programs also by intelligence services.

- ***Browser Hijacker***. A *Browser Hijacker*, also called "hijackware", is a type of malicious program which considerably modifies the behaviour of the victim's web browser. It can be used to make money off unwanted ads, to steal information from users, or to infect the systems with other malware by redirecting users to malicious websites [15].

- ***Bots/Botnet***. In general, *bots* (short for 'robots') are software programs designed to automatically perform specific operations. Some bots are used for legitimate and harmless purposes such as video programming, video gaming, internet auctions and online contest, among other functions. It is however becoming increasingly common to see bots being used maliciously. Malicious bots can be (and usually are) used to form botnets. A botnet is defined as a network of host computers (zombies/bots) that is controlled by an attacker - the *bot-master* [5]. Botnets are frequently used for DDoS (Distributed Denial of Service) attacks, but there are other ways that botnets can be useful to cybercriminals: for example they can be employed to carry out brute force attacks on websites or to distribute malware [10].

- ***Crypto-miner***. Crypto-miners are a relatively new malware family. Cybercriminals employ this type of malicious tools to mine Bitcoin and/or other similar digital currencies on the target machine by exploiting the victim system's computing power, without the owner realising it. Obviously, the mined coins end up in the attackers' digital crypto wallets.

  Recently, a more modern method of crypto-mining that works within browsers (also called crypto-jacking), has become quite popular [16].

- ***File-less malware***. As the term suggests, File-less malware is a type of malware that operates from a victim's computer memory, not from files on the hard drive, taking advantage of legitimate code and tools (known as "LOLBins" [10]) already existing within the system. File-less malware leaves no malware files to scan and no malicious processes to detect, and is therefore harder to detect and remove than traditional malware: it is, in fact, up to ten times more successful [17]. Furthermore, it also renders forensic analysis more difficult because it disappears from the victim's computer upon rebooting.

- ***Keylogger***. Keyloggers, often considered as being a sub-category of spyware, are malicious programs which secretly track keystrokes on a keyboard, without the system's owner consent - action that is usually referred to as *keylogging* or *Keystroke logging*. The collected data is stored and sent to the attacker who uses it to figure out highly sensitive information such as passwords, usernames and payment details. Keylogging can be performed in various ways, ranging from hardware and software-based approaches to the more sophisticated electromagnetic and acoustic analysis [13].

- ***RAM Scraper***. *RAM scraper* malware, also known as *Point-of-Sale (POS)* malware, targets POS systems like cash registers or vendor portals, harvesting data temporarily stored in RAM (Random Access Memory). Doing so the attacker can easily access unencrypted credit card numbers [15].

- ***Ransomware***. *Ransomware*, also known as "encryption" or "crypto" Trojan, is a malicious program that, after having infected a host or network, holds the system captive and displays a message requesting a ransom to the host/network users. In particular, it encrypts data on the infected system (or anyway it locks down the system preventing user access) and only unblocks it when the correct password - decryption key - is entered. Without the latter, is practically impossible to regain access to the system. Digital currencies such as Bitcoin and Ether are the most common means of payment, making it difficult to track cybercriminals. Moreover, paying the ransom does not guarantee the user to receive the necessary decryption key. Additionally, some forms of ransomware threaten victims to publicize sensitive information within the encrypted data in order to convince them to pay the ransom.

- ***Rogue Security Software***. *Rogue Security Software* can be considered as a from of scareware. This type of malware programs presents itself as a security tool to remove risks from the user's system. In reality, this fake security software installs more malware onto the system [15].

- ***Rootkit***. A *rootkit* is generally thought as a type of malicious software, or a collection of software tools, designed to remotely access or control a computer without being detected by users or security programs. An attacker who has installed a rootkit on a system is able to perform a wide range of malicious activities such as remotely executing files, logging user

activities, installing hidden malware, etc. Since a rootkit operates stealthily and continually hides its presence, its prevention, detection and removal can be difficult and often relies manual methods [14].

More recently, the term "rootkit" has often been used to refer also to concealment routines in a malicious program. These highly advanced and complex routines are written to hide malware within legitimate processes on the infected computer. In fact, once a malicious program has been installed on a system, it is essential that it remains hidden, to avoid detection and disinfection [13].

- ***Scareware***. Scareware is a generic term for malware that uses social engineering to frighten and manipulate users, inducing them into thinking their system is vulnerable or has been compromised. However, in reality no danger has actually been detected: it is a scam. The attack succeeds when the user purchases unwanted - and potentially dangerous - software in an attempt to eliminate the "threat". Generally, the suggested software is additional malware or allegedly protective software with no value whatsoever [16].

  Some versions of scareware act as a sort of shadow version of ransomware; they claim to have taken control of the victim's system and demand a ransom. However they are actually just trying to fool the victim [12].

- ***Spyware***. *Spyware*, another name for *privacy-invasive software*, is a type of malicious software that spies on the infected system's user activity. Specifically it can collect various types of personal information about users, such as Internet browsing habits, credit card details and passwords, without their knowledge. The information gathered is then sent back to the responsible cybercriminal(s).

  However, the functions of spyware often go far beyond simple activity monitoring and information gathering. In fact, they may also interfere with the user's control of the computer in other ways, such as installing additional software, redirecting web browser activity and changing computer settings [13].

  Law enforcement, government agencies and information security organizations often use spyware to monitor communications in a sensitive environment or during an investigation [18].

Other cyber-threats

Other cyber threats which are not strictly, and should not be confused with, malware are, for example:

- ***Software Bug***. A software bug is an error, or flaw, in a computer program code or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways. Usually, most of these defects arise from human errors made in the program's source code. Minor bugs only slightly affect the behaviour of a program and can therefore remain undiscovered for quite a long time. On the other hand, more significant bugs can cause crashes or freezes [14].

- ***Software Vulnerability***. In computer security, a vulnerability is a hole or a weakness in an application, which can be a design flaw or an implementation bug, that can be exploited by a threat actor, such as an attacker to bypass access controls such as user authentication, override access privileges, steal data or perform other unauthorized actions within a computer system potentially causing harm to the stakeholders of an application. Stakeholders include the application owner, application users, and other entities that rely on the application [19].

- ***Malvertising***. Malvertising is the use of legitimate ads or ad networks to covertly deliver malware to unsuspecting users' computers. For example, a cybercriminal might pay to place a malicious ad on a legitimate website. When clicked, it either redirects the victims to a malicious website or installs malware on their computer. In some cases, the code embedded in an ad might even execute automatically without any action from the user, a technique referred to as a "drive-by-download".

- ***Phishing***. Phishing is a type of social engineering attack in which the attacker attempts, through email messages (or other means), to trick users into divulging passwords or other personal and financial information, downloading a malicious attachment or visiting a website that installs malware on their systems.

  Some phishing emails are highly sophisticated and can deceive even experienced users, especially if the attacker has successfully compromised a known contact's email account and uses it to spread phishing attacks or malware such as worms. Others are less sophisticated and simply spam as many emails as possible with messages such as "Check your bank account details" [20].

- ***Spam***. In cybersecurity, unsolicited emails are generally referred to as *spam*. Typically, spam includes emails carrying unwanted advertisements, fraud attempts, links to malicious websites or malicious attachments. Most spam emails contain one or more of the following characteristic traits [15]:

  – Poor spelling and grammar
  – Unusual sender address
  – Unrealistic claims
  – Suspicious links

General considerations on malware types

Malware samples are usually categorised both by their means of infection and their behavioural category: for instance, *WannaCry* is a ransomware worm. Moreover, a particular piece of malware may have various forms with different attack vectors: e.g., the banking malware called *Emotet* has been spotted in the wild as both a trojan and a worm [12]. Finally, many instances of malware fit into multiple categories: for example *Stuxnet* is both a worm, a virus and a rootkit.

2.2 Detection evasion

From the creation of the first malwares in 1960s, which were designed by hackers mainly for fun, a strong competition between attackers and defenders has risen. In order to defend from malware attacks, anti-malware groups have been developing increasingly complex (and clever) new techniques. On the other hand, malware developers have conceived and adopted new tactics/methods to avoid the malware detectors [21].

The first types of anti-malware tools were mostly based on the assumption that malware structures do not change appreciably during time. In fact, initially, the malware machine code was completely unprotected. This allowed analysts to exploit opcode sequences to recognise specific malware families. Recently, however, a big advancement led to the so-called "second generation" malware [22], which employs several obfuscation techniques and can create numerous variants of itself, with the purpose of evading such opcode signatures. This posed a challenge to anti-malware developers.

The first malware to exhibit a detection avoidance behaviour was the *Brain* virus [23], in 1986. Such malware, in fact, managed to conceal the infected disk section whenever the user attempted to read it, forcing the computer to display clean data instead of the infected part. From that moment on, the ever increasing popularity of detection evasion techniques among malware writers has shown that malware survival in the victim's machine has become of primal importance: the longer the malware remains undetected, the more harm it can do and the more profitable it is to its writer [5].

2.2.1 Reverse-Engineering

Reverse engineering, in broad terms, indicates the process of extracting knowledge, ideas, design philosophy etc. from anything man-made [24].

Software reverse engineering is the application of reversing methodologies and techniques to extract knowledge from a software product in order to better understand its inner workings.

Both malicious actors and malware analysts/investigators extensively use reversing techniques, but with opposing purposes. Malware developers often use them to discover vulnerabilities in systems or programs, while analysts and antivirus software developers use them mainly for analysing malicious programs to understand how they work, what damages they can cause, how they infect the system and reproduce and, most importantly, how they can be removed, detected and avoided.

2.2.2 Malware analysis

Malware analysis is the process of extracting as much information as possible from malicious samples discovered in the wild, which usually are in the form of machine code executables (compiled executables), in order to determine their purpose and functionality (and associated threats). This process allows security teams to take a number of useful actions such as: developing effective detection techniques against the analysed malicious code, containing its damage, reversing its effects on the targeted system, developing removal tools that can delete it from infected machines (to cleanly remove a piece of malware from an infected machine it is usually not enough to delete the binary itself) and designing methods to guard systems against future infections [25].

Initially malware analysts/researchers had to manually analyse each malware sample. This process was however rather complex, it required high expertise, and was quite time-consuming. Moreover, the number of malware samples that nowadays need to be analysed on a daily basis is of the order of hundreds. This implies that the analysis of malware samples can no longer be exclusively done manually. As such, several analysis tools have been developed in recent years to facilitate analysts in analysing malware samples.

Traditionally, there are two main types of analysis: *static* and *dynamic*. Moreover, these two types can be, and frequently are, combined together (*hybrid* analysis) in various stages of malware analysis to optimize results [5].

Static analysis

Static analysis consists of examining an executable file's code without actually executing it. Static analysis techniques usually extract peculiar features from malicious samples in order to be able to recognise them and distinguish them from benign ones. The features usually extracted are, for example, string signatures, byte-sequence n-grams, library or API calls, opcode frequency distributions, peculiar attributes found in the executable header etc. However, this approach, since it is based on signatures/features extracted from already analysed samples, is not much effective on zero-day and evolutionary malware.

A malware analyst performing manual static analysis usually disassembles the binary first, meaning that he 'translates' the program's machine code instructions back into assembly language, generating a more human-interpretable code listing. The disassembled binary can then be subject of more advanced static analysis techniques such as control flow analysis, data flow analysis and many more. This is done in order to try understanding the program functionality and inner workings, among other useful information [25].

Static analysis has a number of advantages with respect to dynamic analysis, such as that it is usually faster (and safer) than dynamic approaches and that it takes into account the entire program code and not just sub-parts of it. However, a general disadvantage of static analysis is that many times the information collected during this type of analysis is very simple and not always sufficient for a conclusive decision on the malicious intent of a file. It is, however, good practice to start the analysis of a suspicious executable file extracting as much information as possible through various static techniques before passing to the dynamic counterpart. The information statically extracted may in fact provide useful knowledge to better apply dynamic techniques and enhance the final results.

Additionally, another common problem to deal with when using static analysis is that, since malicious code is written directly by the adversary, it can be purposefully designed to be hard to

analyse statically. For example, analysis evasion techniques like packing, encryption and obfuscation can be exploited by malware authors to hinder both disassembly and code analysis steps typical of static analysis approaches, ultimately leading to incorrect or useless information [5].

Dynamic analysis

Contrary to static analysis, *dynamic techniques* analyse the program's code while or after execution in a controlled environment. These techniques, while being non-exhaustive, have the significant advantage that they analyse only those instructions that are actually executed by the running process. This implies that dynamic analysis is less susceptible to anti-analysis attempts like code obfuscation or anti-disassembly [25]. Moreover, dynamic analysis is also more effective in terms of malicious behaviour detection, since it doesn't look at the disassembled code but, through the use of monitoring tools, it tracks the operations that the code performs on the file system, registry, network etc. It is however, computationally more expensive and time consuming.

Basic dynamic analysis consists of observing the sample under analysis interacting with the system. For example, this can be done by first taking a snapshot of the original system state, then introducing the malware into the system, executing it and finally comparing the new system state with the original one. The changes detected can then be used for infection removal on infected systems and/or for modelling effective signatures/features.

Advanced dynamic analysis, on the other hand, consists of directly examining the executed malware internal state while it is being run. This is done typically by monitoring the APIs and OS function calls invoked, the files created and/or deleted, the registry changes and the data processed by the program under analysis during its interaction with the system. The information extracted in this way can be used to understand the malware behaviour and functionality [5].

When using dynamic techniques, however, malware analysts don't simply run malware executables on their own systems, which most probably are connected to Internet, as they could easily escape the analysis environment and infect other hosts/networks. It is, in fact, advised to deploy dynamic techniques on "safe" and controlled (isolated) environments such as dedicated stand-alone hosts, virtual machines or emulators.

The use of clean dedicated hosts, reinstalled after each dynamic analysis run, is however not the most efficient solution due to the environment re-installation process overheads. On the other hand, using virtual machines (for example VMware) to perform dynamic analysis is more efficient. In fact, in this case, since the malware only affects the virtual machine environment, it is enough, after a dynamic analysis run, to simply discard the infected hard disk image and replace it with a clean one. Unfortunately, a significant drawback is that the malware being analysed may determine it is running in a virtualized environment and, as a result, modify its behaviour. To counter this last problem one could make use of emulators, which are theoretically undetectable by analysed malware. These tools, however, run the code under analysis significantly slower and are therefore sometimes detectable using specially crafted time-related code.

Hybrid analysis

Hybrid Analysis is the combination of static and dynamic analysis. It is a technique that integrates run-time information extracted through dynamic analysis with information extracted through static analysis in order to have a complete view of the malware's behaviour while avoiding the problems posed by anti-analysis techniques as much as possible.

2.2.3 Anti-reversing

Anti-reversing techniques were originally meant to complicate the reverse engineering process, making it as difficult as possible for hackers or any malicious user. Attackers employing reversing methodologies can, in fact, get an insight of the logic of the code as well as hidden information by disassembling unprotected files' binaries.

However, anti-reversing techniques are nowadays extensively used also by malware authors in order to make their creations difficult to analyse in an attempt to avoid, or at least postpone, detection as much as possible.

There exist several anti-reversing approaches, each with its own advantages and disadvantages. However it is common practice to use a combination of more than one of them for protecting the same file.

In the next sections some of the more common anti-reversing techniques are discussed.

2.2.4 Anti-disassembly

Anti-disassembly techniques use specially crafted code and/or data in a program to cause disassembly analysis tools to generate an incorrect program listing [26]. The attackers' usage of these techniques thus implies a time-consuming analysis for malware analysts, ultimately preventing the retrieval/reconstruction of the source code in a reasonable time. Furthermore, anti-disassembly techniques may also inhibit various automated analysis tools and heuristic-based engines which take advantage of disassembly analysis to identify or classify malware.

These techniques exploit the disassemblers' inherent weaknesses and assumptions about the code being analysed, which open an opportunity for malware authors to deceive the analysis.

For example, while disassembling a program, each sequence of executable code instructions can have multiple disassembly representations, some of which may be invalid and obscure the real purposes of the program. Therefore, malware authors, in order to add anti-disassembly functionality to their creations, can produce sequences of code that deceive the disassembler into outputting a list of instructions that differs from those that would be executed at runtime [26].

There are two types of disassembler algorithms: linear and flow-oriented (recursive). In particular, the linear variant is easier to implement, but it is also more simplistic and error-prone with respect to the other one.

Linear Disassemblers

The *linear* disassembly strategy is based upon the basic assumption that the program's instructions are organized one after the other, linearly. In fact, this type of disassemblers iterates over a block of code, disassembling one instruction at a time, sequentially, without deviating. More specifically, the tool uses the size of the currently disassembled instruction to figure out what bytes to disassemble next, without accounting for control-flow instructions [26].

Linear disassemblers are therefore easy to implement and work reasonably well when working with small sections of code. They introduce, however, occasional errors even with non-malicious binaries. In fact, one of the main drawbacks of this technique is that it blindly disassembles code until the end of the section, assuming the bytes encountered are nothing but instructions packed together, without distinguishing between code, data and pointers.

In a PE-formatted executable file, for example, the executable code is typically contained inside a single ".text" section. However, for almost all binaries, this code section contains also data, such as pointer values. These pointers are blindly disassembled and interpreted by the linear disassembler as instructions. Malware authors can therefore exploit this behaviour for example by implanting in the code section data bytes that form the opcodes of multi-byte instructions.

Flow-Oriented Disassemblers

The *flow-oriented* (or *recursive*) disassembly strategy is more advanced than the previous one and is, in fact, the one used by most commercial disassemblers like *IDA Pro* [26].

Differently form the linear strategy, the flow oriented one examines each instruction, builds a list of locations to disassemble (the ones reached by code) and keeps track of the code flow. This implies that, while disassembling a code section, this type of disassembler will not blindly parse

the bytes immediately following the JMP instructions' ones, but it will disassemble the bytes at the jump destination addresses.

This behaviour is more resilient and generally provides better results with respect to the linear approach, but also implies a greater complexity. In fact, while a linear disassembler has no choices to make about which instructions to disassemble at any given time, flow-oriented disassemblers have to make choices and assumptions, in particular when dealing with conditional branches and call instructions.In particular, in the case of conditional branches, the disassembler needs to follow both the false branch (most flow-oriented disassemblers will process the false branch of any conditional jump first) and the true one. In typical compiler-generated code there would be no difference in output if the disassembler processes one branch or the other first. However, in handwritten assembly and anti-disassembly code, taking one branch or the other first can often produce different disassembly listings for the same block of code, leading to problems in analysis.

Anti-Disassembly Techniques

Jump Instructions with the Same Target One of the most used anti-disassembly techniques consists of two consecutive conditional *jump* instructions both pointing to the same target [26].

Here is an example:

```
1 74 03   jz  loc
2 75 01   jnz loc
3
4 loc:
```

Listing 2.1: Jump Instructions with the Same Target

In this case, the conditional jump '**jz loc**' is immediately followed by a jump to the same target but with opposite condition: '**jnz loc**'. This implies that the location **loc** will always be jumped to. Consequently, the combination of **jz** with **jnz** acts, in this case, like an unconditional **jmp** instruction. A disassembler, however, won't recognize this given that it disassembles just one instruction at a time. During the disassembly process, in fact, when the **jnz** instruction is encountered, the disassembler takes the false branch of the instruction first and therefore continues disassembling, even though this branch will never be executed in practice.

Jump Instructions with a Constant Condition Another common anti-disassembly technique is composed of a single conditional *jump* instruction with an always true (or false) condition [26].

Example:

```
1 33 C0   xor eax, eax
2 74 01   jz  loc
3
4 loc:
```

Listing 2.2: Jump Instructions with a Constant Condition example

The first instruction in the example code, **xor eax, eax**, sets the **EAX** register to zero and, consequently, it sets the zero flag. The next instruction, **jz** (jump if zero flag is set), appears to be a conditional jump but in reality is not conditional at all. In fact the the zero flag will always be set at this point in the program execution. The disassembler, however, will process the false branch first, even if in reality it would never trigger.

Impossible Disassembly The simple anti-disassembly techniques mentioned above are frequently coupled with the use of a, so called, *rogue byte*. A *rogue byte* is a data byte, which is not part of the program logic flow, strategically placed after a conditional *jump* instruction in order to trick the disassembler. The byte inserted usually is the opcode for a multi-byte instruction, therefore disassembling it prevents the real following instruction from being properly disassembled [26].

In all these cases, however, a reverse engineer is able to properly disassemble the code with the use of interactive disassemblers like IDA Pro, ignoring the *rogue bytes*. However, there are some conditions in which no traditional assembly listing can accurately represent the instructions that are executed. These are exploited by the so called *impossible disassembly* techniques which produce code that can be disassembled only by using a vastly different representation of the code than what is provided by currently available disassemblers.

The core idea behind these techniques is to make the *rogue byte* part of a legitimate instruction that is executed at runtime. This way the *rogue byte* becomes not ignorable during disassembly. In this scenario any given byte may be a part of multiple instructions that are executed. This is done using *jump* instructions. The processor, while running the code, will interpret and execute the bytes following the logical flow of the program, so there is no limitation on the number of instructions the same byte can be part of; a disassembler, however, has such limitations since it will usually represent a single byte as being part of a single instruction.

Example:

```
1  EB
2        JMP -1
3  FF
4        INC EAX
5  C0
6  48    DEC EAX
```

Listing 2.3: Impossible Disassembly example

In this simple example the first instruction is a 2-byte *jmp -1* instruction (**EB FF**). Its target is its own second byte. At run time this causes no errors because the **FF** byte is the first byte of the next instruction *inc eax* (**FF C0**). However, when disassembling, if the disassembler interprets the **FF** byte as part of the *jmp* instruction, it won't be able to interpret it also as the beginning of the *inc eax* instruction.

This 4-byte example code increments the **EAX** register, and then decrements it, therefore it is essentially a complex **NOP** sequence. Being a simple, and small, sequence it could easily be inserted at any location in a program's code in order to fool disassemblers. However this sequence is also easily recognisable by reverse engineers and substituted with **NOP** instructions using IDA Pro or other instruments and/or scripts or forced to be interpreted as data bytes and therefore skipped by the disassembler.

However this was only a simple example sequence. More complex and ingenious sequences can be made to fool disassemblers while being harder to detect.

Control Flow Obscuring Techniques

Control-flow analysis (CFA) is a static-code-analysis technique for determining the control flow of a program. Modern disassemblers like IDA Pro are able to correlate function calls and extract high-level information about the program knowing how functions are related to each other [26].

Control-flow analysis can however be easily defeated by malware authors by employing simple but effective tricks.

The Function Pointer Problem Function pointers are a common programming idiom present in widely adopted programming languages such as **C**, while being extensively used in the background in object oriented languages like **C++** and **Java** [26].

As opposed to referencing a data value, a function pointer points to executable code within memory. Dereferencing the function pointer yields the referenced function, which can be invoked and passed arguments to as in a normal function call. Such invocation is also known as an 'indirect' call, since, doing so, the function is indirectly invoked through a variable instead of directly through a fixed identifier or address. In assembly code this corresponds to a *call* instruction with a function pointer as argument.

Function pointers, however, greatly reduce the information that can be automatically extracted by disassemblers about the program control flow. Moreover, if function pointers are used in specially crafted, or non-standard code, the resulting executable can be difficult to reverse-engineer without the use of dynamic analysis techniques.

As a result, function pointers, in combination with other anti-disassembly techniques, can greatly increase the complexity and difficulty of reverse-engineering.

Return Pointer Abuse Among the instructions capable of transferring control within a program the most obvious ones are the already mentioned *call* and *jmp* instructions; however there are other more subtle ways malware authors can change the control flow of a program [26]. One example to this is the explicit use of the *retn* instruction.

The *retn* instruction is generally used in combination with the *call* instruction to properly return from the called subroutine/function: when a *call* instruction is reached during program execution, a return pointer, which points to the address of the instruction immediately following the end of the *call* instruction itself, is pushed on the stack, before jumping to the call instruction target. The pushed return address is then used upon reaching the *retn* instruction to exit the current subroutine and return to the calling one.

The *call* instruction can be thus seen as the combination of a *push* and *jmp*; the *retn* instruction, on the other hand, is the combination of *pop* and *jmp*: it pops the last value pushed to the stack and jumps to it.

The *retn* instruction is therefore typically used to return from a function call, but it could also be used for other purposes. When used for such other reasons the disassembler is generally fooled, because it still will interpret it as a return from a function call. Therefore it won't show any code cross-reference to the target being jumped to and will also prematurely terminate the function being analysed.

Misusing Structured Exception Handlers Another powerful anti-disassembly (and anti-debugging) technique works by exploiting the Structured Exception Handling (**SEH**) mechanism, which is extensively used by *C++* and other programming languages since it provides programs a way to smartly handle error conditions [26].

Exceptions can be triggered for various reasons: for example when dividing by zero or accessing an invalid memory region. Moreover, software exception can also be raised by the code itself by calling the *RaiseException* function. When an exception is raised it makes its way through the **SEH** chain, which is a list of functions specifically designed to handle exceptions, until it is caught by one exception handler in the chain. Each function in the list can either handle the exception (a.k.a. *catch* it) or pass it to the next handler in the list. *Unhandled exceptions* are the ones that make their way to the last handler. The last exception handler is the code responsible for triggering the 'unhandled exception' message to the user.

Exception handling is used in almost all programs and exceptions happen regularly in most processes (and are handled silently). A malicious actor could, however, exploit this mechanism to achieve covert flow control by adding his own specially crafted handler on top of the **SEH** chain.

This can be done at runtime simply by pushing some specific values on the stack, effectively adding a new entry in the Exception handling chain. This procedure, however, is subject to the constraints imposed by the Software Data Execution Prevention (**Software DEP**), which is a security feature that prevents the addition of third-party exception handlers at runtime. However various workarounds to this protection can be used in the case of handwritten assembly code.

2.2.5 Anti-debugging

Another popular anti-analysis technique, besides anti-disassembly, is *anti-debugging*. Malware authors use anti-debugging techniques to recognise when their malicious program is under the control of a debugger or to interfere with the debugger behaviour. This is done in an attempt

to slow down the malware analysts who use debuggers to understand how the malware operates. Malwares using these techniques usually alter their normal control flow paths or cause crashes if they detect they are running in a debugger, thus interfering with analysis [26].

Windows Debugger Detection

In Windows OS various techniques can be used to detect if a process is being run in a debugger: for example this can be done by exploiting the Windows API itself or by manually checking memory structures looking for debugger artefacts [26].

Using the Windows API One of the simplest ways to know if a debugger is attached to a process is by using Windows API functions. Inside the Windows API there are, in fact, functions that were specifically designed to detect debuggers; moreover some functions that were originally created with other purposes can also be used for performing debugger detection [26].

Malware analysts can counter this technique by manually modifying the malware code during execution, in particular the API function result flag, after the call to make sure the desired path is taken, or by straight up removing/skipping the API function call.

Here are some examples of common Windows API functions used for *anti-debugging* purposes:

- **IsDebuggerPresent** This is the simplest API function that can be used for detecting a debugger. It determines whether the **current** process is being debugged by a user-mode debugger. It does so by getting the value of the field *IsDebugged* from the Process Environment Block (**PEB**) structure. In particular this functions returns zero if the process is not running within a debugger context and a non-zero value otherwise.
- **CheckRemoteDebuggerPresent** This API function is similar to the previously described one (*IsDebuggerPresent*) but it checks for a 'remote' debugger on the specified process. The term 'remote' in the name CheckRemoteDebuggerPresent does not imply that the debugger necessarily resides on a different machine; instead, it indicates that the debugger resides in a separate and parallel process. This function takes a process handle as argument, and checks if that process has a debugger attached. It can however be used also to check the current process by passing its handle.
- **NtQueryInformationProcess** This function can retrieve different kinds of information from a process. The first argument for this function is the process handle, the second one is the ProcessInformationClass parameter which specifies the information to retrieve. When using the value *ProcessDebugPort* for this parameter, for example, the function returns a zero if the process is not being debugged; otherwise a non-zero value representing the debugger port number is returned.
- **OutputDebugString** This function, originally designed to simply send a string to a debugger for display, can also be used to detect the presence of a debugger. In fact, in there is no debugger attached, the function will internally set the last-error code. In a few lines of code it is thus possible to know if a debugger is present or not:

```
DWORD errorValue = 12345;
// set custom last error code
SetLastError(errorValue);

// try outputting string on debugger;
// if no debugger is present, it will set
// the last-error code to a new value
OutputDebugString("Test for Debugger");

if(GetLastError() == errorValue){
  // a debugger is present
  ExitProcess();
}
else{
  // no debugger was detected
```

```
16      RunMaliciousPayload();
17  }
18
```

Listing 2.4: OutputDebugString debugger detection

Manually Checking Structures Malware authors usually don't simply take advantage of the Windows API functions for detecting the presence of a debugger, rather they generally prefer checking the PEB structure (and others) by themselves. One of the reasons why this is true is that API calls can be easily hooked by a rootkit to return false information, thus thwarting this technique [26].

- **Checking the BeingDebugged Flag** The Windows PEB structure contains all user-mode parameters associated with a process, including the process's environment data such as environment variables, addresses in memory and debugger status, among other things.

  A malicious program can explicitly check the *BeingDebugged* flag within the PEB structure to understand if a debugger is attached to its process (it is zero if no debugger is present).

  Example:

```
1  mov   eax,  dword ptr fs:[30h]  ; get PEB address
2  mov   ebx,  byte ptr [eax+2]    ; get BeingDebugged flag value
3  test  ebx,  ebx                 ; test if the value is 0
4  jz    NoDebuggerDetected        ; if 0, no debugger was detected
5
```

  Listing 2.5: BeingDebugged manual check

  Malware analysts can counter this technique by detecting this specific code sequence (and other similar ones) in the code and then manually changing the *BeingDebugged* flag to zero, or alternatively forcing the jump to be taken (or not) by modifying the zero flag before the jump instruction.

- **Checking the ProcessHeap Flag** The *ProcessHeap*, which is an undocumented location within a reserved array inside the PEB structure, contains the location of the first heap of a process allocated by the loader. This heap can be used for debugger detection purposes since it contains some information telling if it was created within a debugger or not. In particular malware usually check the values of the fields called *ForceFlags* and *Flags*.

  To overcome this technique, malware analysts can change the *ProcessHeap* flags manually or use a *hide-debug* plug-in for their debugger.

- **Checking NTGlobalFlag** Processes started within a debugger run slightly differently than others, therefore they create memory heaps differently. The information needed to determine how to create heap structures is stored at an undocumented location in the PEB. Practically, a value of *0x70* at this location indicates that the process is running within a debugger.

  Again, in order to counter this technique, malware analysts can change the flags manually or use a *hide-debug* plug-in for their debugger.

Checking for System Residue Debugging tools typically leave some traces of their presence on the system. Malicious programs can therefore be designed to search for these traces in order to determine when it is being analysed. For example, malware can search for references to debuggers in the registry keys [26]. Moreover, malware can also be designed to search through the system for files and directories commonly related to debuggers, such as debugger program executables. Finally, malwares can also detect debugger residues in live memory, by viewing the current process listing or, more commonly, by performing a *FindWindow* in search for a debugger.

Identifying Debugger Behaviour

Debuggers are very useful to malware analysts because they can be used to set breakpoints in the code or even to single-step through a process running code to ease the reverse-engineering process. These operations, however, modify the process code and are therefore easily detectable [26].

INT Scanning A common anti-debugging technique exploited by malware authors consists in making the process scan its own code in search for an **INT 3** (opcode *0xCC*) instruction. **INT 3** is, in fact, a software interrupt used by debuggers: when setting a breakpoint the debugger replaces the target instruction in the running program with the opcode *0xCC* (INT 3) which causes the process to call the debug exception handler [26].

Malware analysts can counter this technique exploiting hardware breakpoints instead of software ones.

Performing Code Checksums Another anti-debugging technique consists in calculating the checksum (cyclic redundancy check (CRC) or MD5 checksum) of a section of the process' own code. This has the same net effects as scanning the code for software interrupts [26].

Again this technique can be countered by using hardware breakpoints instead of software ones, or by modifying the program's control flow at runtime with a debugger.

Timing Checks One of the most widespread techniques for debugger detection is to perform *timing checks*. Processes, in fact, tend to run substantially slower when executed within a debugger context. Moreover analysts usually run programs in single steps in order to better understand the code behaviour; this in turn greatly increases execution time [26].

Using timing checks it is possible to detect a debugger in different ways:

1. Recording one timestamp before and another after the execution of some operations and then comparing them. If the lag is greater than a specified threshold then a debugger is probably being used.

2. Recording 2 timestamps, one before and the other after raising an exception. If the current process is being debugged then the exception will be handled by the debugger itself more slowly than normal. Moreover, by default, debuggers ask for human intervention when an exception occurs, thus causing huge delays.

Typically, the timestamps are recorded using directives such as the *rdtsc* Instruction and the *QueryPerformanceCounter* and *GetTickCount* windows API functions. In particular, the *rdtsc* instruction and the *GetTickCounter* return the number of ticks/milliseconds that have elapsed since the last system reboot while the *QueryPerformanceCounter* queries a high-resolution counter which stores counts of activities performed by the processor. These functions, when used as described above, allow malwares to detect the presence of debuggers.

The use of timing checks for anti-debugging purposes can be discovered by malware analysts either during debugging or while performing static analysis by identifying specific sequences of instructions. Moreover, these timing checks more frequently detect debuggers when the analyst is single-stepping though the code or setting a breakpoint between the two time related instruction calls. This implies that, to counter this technique, malware analysts can avoid setting breakpoints and single-stepping in the protected code regions, or alternatively modify the result of the timestamps comparison as needed.

2.2.6 Anti-virtual machine

Malware analysts often use virtual machines (VMs) or other isolated environments like sandboxes, to analyse malware samples. With the purpose of evading analysis and bypassing security systems, malware authors often design their code to detect isolated environments. The techniques used with such purpose are called *Anti-virtual machine* techniques (Anti-VM). Once a virtual machine is detected, the evasion mechanism may alter the malware's behaviour, or it may even prevent the malicious code from running altogether [26].

VMware Artefacts

Virtual machines are designed to emulate real hardware functionality. To achieve that, however, some artefacts inevitably remain on the system, which can reveal that a virtual machine is indeed being used. These kinds of artefacts can be specific files, processes, registry keys, services, network device adapters etc. [26].

Here are some examples of anti-virtual machine techniques applied to detect VMware virtualization software:

- **Checking for Processes Indicating a VM**. When a *VMware* virtual machine is running and *VMware tools* is installed, three VMware-related processes can be found in the system process listing: *VMwareService.exe*, *VMwareTray.exe* and *VMwareUser.exe*. A malicious software can therefore easily detect if VMware is being run searching through the process listing for the *VMware* string.
- **Checking for Existence of Files Indicating a VM**. The VMware default installation path usually also contains artefacts. Searching for the string *VMware* in such location may reveal the use of a virtualized environment.
- **Checking for Registry Keys**. *VMware Tools* may leave some artefacts also in the registry. More specifically, the presence of specific registry entries may reveal the use of VMware.
- **Checking for Known Mac Addresses**. In order to connect a virtual machine to a network it needs to have its own virtual network interface card (NIC). This implies that VMware software needs to create a MAC address for the virtual machine, to associate to its NIC. However, VMware utilises default addresses with a specific starting sequence which depends on its current version. Therefore a malicious software may be able to identify it is running in a VMware virtual machine simply by checking the system MAC address against common VMware values.

In order to counter anti-virtual machine techniques, malware analysts need to identify the specific checks for VMware artefacts in the malicious code and then manually patch them. For example, depending on the anti-VM technique used, they may patch the malware code while debugging to artificially make all checks pass, or modify the name of VMware processes in order to make them undetectable by the malicious software.

Vulnerable Instructions

The *virtual machine monitor program*, which monitors the virtual machine execution, has some security weaknesses that may allow malware to detect its usage. In particular, in order to avoid performance issues deriving from fully emulating all instructions, VMware allows certain instructions to execute without being properly virtualized. This in turn means that certain instruction sequences may return different results when running within a VMware virtualized environment than they do on native hardware. This discrepancy can be used by malware authors to detect VMware usage [26].

However, those specific instructions are not typically used within a program unless it is specifically performing VMware detection, because they are useless when executed in user mode. Therefore, avoiding this type of anti-VM technique can be as easy as patching the malicious code to prevent it calling these instructions.

2.2.7 Packers and unpacking

Packing programs, commonly known as *packers*, are software programs that take an *executable file* or *dynamic link library (DLL)*, compresses and/or encrypts its contents and then packs it into a new executable file [26].

When packers are used on malicious programs, the malicious code appearance is changed as a consequence of the applied compression and/or encryption. The packing of files thus hinders basic static analysis and simple malware detection techniques. Moreover, a packer specifically designed to make the file difficult to analyse may even employ anti-reverse-engineering techniques, such as anti-disassembly, anti-debugging and/or anti-VM on the resulting compressed version; on top of that some packers, using randomization, are also able to generate different variants of a single file every time it is packed [27].

Malware authors have therefore increasingly been using these tools to hide their creations from anti-malware solutions and malware analysts. In order to analyse packed malware, in fact, it must be unpacked first. Properly unpacking a packed program, however, is generally not easy.

Packed files usually contain two basic components:

- A number of data blocks containing the compressed and/or encrypted version of the original executable file.
- An unpacking stub able to dynamically recover the original executable file at runtime.

When the packed file is executed, the unpacking stub is loaded by the OS and begins unpacking the original executable code in memory. When the unpacking process is completed the control flow is transferred, with a *jmp*, *call* or the more stealthy *retn* instruction (also referred to as '*tail jump*)', to the original file entry point (OEP). This implies that someone attempting to perform static analysis on the packed program, would actually analyse the unpacking stub and not the original code.

Packer types

Commercial and custom made packers can be divided in several levels of complexity depending on the packing techniques used and the additional features they have. The authors of [28] identified 6 main types of packers with increasing complexity. Packer types from 1 to 5 allow, sooner or later at runtime, to have a complete view over the original (unpacked) malicious code, meaning that the unpacker stub unpacks all the code at once. However, what makes them differ is the amount and complexity of the encryption (and obfuscation) methodologies used during packing. On the other hand, type 6 packers unpack only a slice of code at a time in memory, never revealing the whole original code altogether. This implies that malware analysts need to take several memory dumps, instead of just one, if they want to get the complete unpacked code.

Another possible classification of packers can be made based on their purposes and behaviours. Following this idea packers can be broadly classified into the following four categories [29]:

- **Compressors** utilise compression algorithms to shrink files while exploiting few or no anti-unpacking tricks. Popular compressors include the Ultimate PE Packer (UPack), Ultimate Packer for Executables (UPX), and ASPack.
- **Crypters** encrypt and obfuscate the original file contents. Crypters such as Yoda's Crypter and PolyCrypt PE are widely used among malware developers.
- **Protectors** combine features from both compressors and crypters. Some popular commercial protectors are Armadillo and Themida.
- **Bundlers** are used to pack a software package of multiple executable files into a single bundled executable. The files within the package can then be unpacked and accessed without extracting them to disk. Some notable PE bundlers are PEBundle and MoleBox.

Packers detection

Packed executables can be detected through a heuristic approach known as *Shannon Entropy Calculation*. Entropy is, generally speaking, a measure of uncertainty, disorder, in a system or program. The idea behind this approach is that compressed or encrypted executables tend to resemble random data, thus they have higher entropy than unencrypted/uncompressed programs. This approach, however, does not tell any information about the packer used to obtain the packed sample [26].

One common way to tackle this problem is through packer signatures checking. Tools like PEiD and Sigbuster use such method. These tools are, however, not always successful due to the huge number of packer variations and evolutions present in the wild, and the fact that malware authors frequently modify commercially available packers code or create their own packers so that their packed malicious programs do not match any known signature.

Unpacking

Unpacking is the process of restoring the original contents from packed executables in order to allow AV programs and security researchers to analyse the original executable code. There are three different techniques to unpack a packed executable: *automated static unpacking*, *automated dynamic unpacking* and *manual unpacking* ([26], [29]).

Automated static unpacking programs are dedicated routines designed to decompress and/or decrypt executables packed by specific packers, without actually executing the suspicious programs. This method, when it works, is the fastest and most secure method to unpack an executable. Automatic static unpackers are, however, specific to a single packer. Moreover, they are not able to unpack packed samples that were created with the intention of hindering analysis.

Automated dynamic unpackers, on the other hand, use programs to run or emulate the packed executable allowing the unpacking stub to unpack the original executable code in memory. Once the original executable is unpacked, the in-memory program's code is written on disk, and the automated unpacker reconstructs the original import table.

Most often security researchers prefer to perform manual unpacking. The two most common approaches used to manually unpack a program are:

- Discover what packing algorithm has been used to pack a sample and then write a custom program/script to revert it. This process is however quite time consuming.
- Manually run the packed program to allow the unpacking stub to unpack the original code in memory, then dump the process on disk and finally manually modify the PE header so that the program is complete. This process is more efficient than the previous one.

2.2.8 Code Obfuscation

Obfuscation is a technique that generally makes programs harder to understand [30], both for humans and automatic tools. In order to do so, it transforms a program into a new version which is structurally different and more difficult to analyse while retaining the same functionality as the original. The new version of the program is therefore said to be *computationally equivalent* to the original one [31].

Originally, this technology was conceived for the legitimate purpose of protecting the intellectual property of software developers; however it has been widely exploited also by malware authors to evade detection [32]. In particular, in order to elude anti-malware scanners, malware typically use obfuscation techniques to evolve their body into new generations [33], which eventually can be even harder to disassemble and analyse.

Obfuscation techniques can be broadly divided into 2 main sub-categories:

- *Data-based* obfuscation

- *Control-based* obfuscation

However, malware authors usually combine those 2 types of obfuscation techniques in complex and ingenious ways to strengthen the resulting protection [34].

Data-Based Obfuscation

Data-based obfuscation techniques focus on modifying data values and non-control computations. In the following paragraphs some of the most common data-based obfuscation techniques will be described.

Constant Unfolding *Constant folding* is a technique commonly used by compilers to optimize a program's code. It does so by replacing expressions with results known at compile time with the results themselves [34].

For example, a compiler usually transforms the following expression 2.6, into 2.7.

```
x = 4 * 5;
```

Listing 2.6: Before constant folding

```
x = 20;
```

Listing 2.7: After constant folding

Constant unfolding is, instead, an obfuscation technique that performs the exact inverse operation: it replaces the constants in the program's code with some expressions having the constant as a result.

For example, the listing 2.8, after *Constant unfolding* may become 2.9. The two listings are equivalent. Moreover, there is an infinite amount of listings equivalent to 2.8 that can be generated following this principle.

```
push  0h
```

Listing 2.8: Before constant unfolding

```
push  0F9CBE47Ah
add   dword ptr [esp],  6341B86h
```

Listing 2.9: After constant unfolding

Data-Encoding Schemes The previously described technique is, however, easily defeated by simply applying the standard compiler's *constant folding* optimization. This is possible because both the data encoding and decoding functions ($f(x) = x - 6341B86h$ and $f_{-1}(x) = x + 6341B86h$ respectively) were present in the code one after the other. To prevent this flaw, *fully Homomorphic* (operation-preserving) mappings can be employed, together with the application of some operations on the encoded data before decoding it back. This technique is however still not widely used because the automatic tools that apply it are still too inefficient [34].

Dead Code Insertion *Dead code elimination* is another common compiler optimization technique. Its objective is to remove program statements/expressions that nave no real effects on the program operation and final results [34].

For example, the listing 2.10, using *dead code elimination* would become 2.11.

```
int f(){
  int x, y;
  x = 1;    // this assignement is useless, here x is dead
  y = 2;    // y is never used, it is thus dead.
  x = 3;
  return x; // x is live
}
```

Listing 2.10: Before dead code elimination

```
int f(){
  return 3;
}
```

Listing 2.11: After dead code elimination

Obfuscators, on the other hand, use the so-called *dead code insertion* technique in an attempt to make the code harder to follow. This technique performs the inverse operation with respect to *dead code elimination*, adding dead code in the original program's code. However, when used alone, this techniques produces an obfuscated program that can be efficiently de-obfuscated by using the compiler's dead code elimination optimization.

Arithmetic Substitution via Identities This technique aims at replacing certain operators with combinations of other operators with equal net result. By exploiting the equivalence between different combinations of operators the code can be changed arbitrarily without modifying the effective program behaviour and final result [34]. Here are some examples of operator equivalences:

```
-x == !x + 1

x-1 == !(-x)

x+1 == -(!x)

rotate_left(x,y) == (x << y) | (x >> (bits(x) - y))

rotate_right(x,y) == (x >> y) | (x << (bits(x) - y))
```

Listing 2.12: Operators equivalences

Register Reassignment Another simple obfuscation technique is called *register reassignment*. An obfuscator using this technique switches the registers used throughout the code at every application, while keeping the same program code and behaviour [33]. An analyst/attacker using wildcard searching, however, easily defeats this technique.

Instruction Substitution *Instruction substitution* creates variants of a program's original code by replacing some instructions with other equivalent ones [33].

Pattern-Based Obfuscation *Pattern-based* obfuscation is another commonly used technique similar in principle to *instruction substitution*, but more complex. It consists in constructing *patterns* (transformations) that map single or multiple adjacent instructions into a more complex, computationally equivalent, sequence of instructions [34].

For example, the sequence in listing 2.13 might be converted into the one in 2.14, as well as 2.15 or even 2.16.

```
push  reg32
```

Listing 2.13: Original sequence

```
push  imm32
mov   dword ptr [esp],  reg32
```

Listing 2.14: Obfuscation using pattern 1

```
lea   esp,  [esp-4]
mov   dword ptr [esp],  reg32
```

Listing 2.15: Obfuscation using pattern 2

```
sub   esp,  4
mov   dword ptr [esp],  reg32
```

Listing 2.16: Obfuscation using pattern 3

Moreover, patterns can be arbitrarily complicated. For example a listing such as 2.17, could be substituted by the more complex 2.18.

```
1 sub   esp,  4
```

Listing 2.17: Original sequence #2

```
1 push  reg32
2 mov   reg32,  esp
3 xchg  [esp],  reg32
4 pop   esp
```

Listing 2.18: Obfuscation of sequence #2

Malware authors (and also software developers wishing to protect their intellectual property) can use hundreds of patterns in the same program. Moreover, most protections randomly apply patterns so that obfuscating the same program multiple times yields different results. On top of that, patterns can also be applied iteratively: after transforming the original code **C** into **C'** using pattern **P**, another pattern **P'** can be applied to **C'** in order to obtain **C"**, and so on.

Some patterns preserve *semantic equivalence*, meaning that the CPU state will be the same when executing them or the original code. Some other patterns, however, do not. Therefore, depending on the code logic, some substitutions are safe (meaning that the program behaviour and final results are preserved) while others are not. This makes the job of an obfuscator challenging.

Control-Based Obfuscation

Standard static analysis tools generally make assumptions similar to the ones human reverse engineers make when analysing code. Compilers, in fact, predictably translate control flow constructs and data structures. As a result, reverse engineers (and static analysis tools) can easily recognise the original code high level control flow. *Control-based obfuscation* transforms the code control flow structures in non standard ways in order to complicate both static and dynamic code analysis [34].

Some examples of standard static analysis tools assumptions are:

- The *CALL* instruction is always used with the sole purpose of invoking functions.
- Both sides of a conditional branch may feasibly be taken at runtime.
- Function calls almost always return.
- All control transfers target code locations, not data locations.
- Exceptions are used in standard and predictable ways.
- etc..

By violating these assumptions, *control-based obfuscation* techniques confuse disassemblers and other static analysis tools making the analysis more difficult.

Functions In/Out-Lining Reverse engineers frequently rely on control-flow and call graphs to better understand a program's high-level logic. In particular a call graph represents calling relationships between subroutines (functions) in a computer program. Each node of a call graph represents a procedure and each edge ($\mathbf{f} \rightarrow \mathbf{g}$) indicates that function **f** eventually calls procedure **g**. By making the call graph harder to interpret, obfuscators can hinder the reverse engineers' capability of understanding the program behaviour [34]. In order to do so, following two techniques can be used:

- **Inline functions**. The code belonging to a sub-routine is merged into the code of its caller. If the sub-routine is called multiple times, however, the code size can quickly grow.

- **Outline functions**. A subpart of a function is extracted and transformed into an independent function and replaced by a call to the newly created function.

Using these two operations in combination on a program's code results in a degenerated call graph with no clear logic. Moreover, also the functions' prototypes can be modified by adding extra fake arguments, reordering them and so on, to further hide the high-level logic.

Destruction of Sequential and Temporal Locality Usually, in non-obfuscated code, the instructions of a single basic block lie one after the other (*sequential locality*), and basic blocks related to one another (such as successive blocks) are close to each other (*sequential locality of temporally related code*). This is done in order to maximize the instruction cache locality and reduce the number of branches in the final code. Reverse engineers thus can usually rely on the fact that all the code responsible for a specific operation resides in a single region [34].

Malware authors can therefore make manual analysis more difficult by violating this assumption with the introduction of unconditional branches that break sequential locality and temporal locality of multiple basic blocks. However, by constructing the control-flow graph and removing spurious unconditional branches the original control flow can be restored.

Processor-Based Control Indirection Instructions like *JMP* and *CALL* are, for most processors, the 2 essential control flow transfer primitives. In order to make analysis more difficult, one could obfuscate these primitives, for example by using dynamically computed branch addresses or by emulating them.

For example the *JMP* instruction 2.19, can be replaced by the (almost) semantically equivalent listing 2.20.

```
1  jmp   target_addr
```

Listing 2.19: Processor-based control indirection, before

```
1  push  target_addr
2  ret
```

Listing 2.20: Processor-based control indirection, after

Operating System-Based Control Indirection Obfuscation techniques can also exploit operating system primitives and structures similarly to the previously mentioned anti-disassembler techniques. For example, the Structured Exception Handler (*SEH*), Vectored Exception Handler (*VEH*) and Unhandled Exception Handler are commonly used to obfuscate the control flow of Microsoft Windows executables (in Unix-like systems the signal handlers *setjmp* and *longjmp* are commonly used instead) [34].

Subroutine Reordering *Subroutine reordering* is an obfuscation technique that randomly changes the order of a program's subroutines in the original code. This technique can thus generate $n!$ code variations, where n is the number of subroutines [33].

Opaque Predicated An *opaque predicate* is a non-trivial boolean expressions with a constant result (always true or always false) known only at compilation/obfuscation time. Combining it with a conditional *jmp* instruction introduces an additional branch in the control flow graph (*CFG*). This specific combination corresponds to the previously mentioned *Jump instruction with a constant condition* anti-disassembly technique. The added branch should look as real as possible in order to elude detection, and it can be used to insert junk code or to form cycles in the control-flow graph to better hide the original program's logic [34].

Simultaneous Control-Flow and Data-Flow Obfuscation

Data-flow obfuscation and *Control-flow obfuscation* techniques are commonly used together to complicate analysis.

Junk Code Insertion This technique consists in introducing a dead code block (meaning that it will never be executed at runtime) between two other code blocks. Typically utilised in conjunction with *opaque predicates*, this technique is used to further confuse a disassembler that is disassembling an invalid path. Moreover, the junk code typically contains partially invalid instructions, or branches to invalid addresses with the objective of over-complicating the CFG [34].

```
1  push  eax
2  xor   eax,  eax
3  jz    9
4  ;<junk code start>
5  jg    4
6  inc   esp
7  ret
8  ;<junk code end>
9  pop   eax
```

Listing 2.21: Junk Code example

Listing 2.21 presents an example of this technique. More precisely the instruction at line 2 (*xor eax, eax*) zeroes the *EAX* register clearing the zero flag (it is set to 0); therefore, at runtime, the conditional jump (*jz 9*) at line 3 is always taken. The following instructions are therefore junk code.

Control-Flow Graph Flattening *Control-flow graph flattening* consists in replacing all control structures within a sub-part of the control flow graph with a single switch statement commonly called *dispatcher*. This is done with the purpose of hiding the true basic block relationships within the dispatcher. When using this technique, first a subpart of the program's control flow graph is selected to be substituted by the dispatcher. Some transformations may then be applied to the basic blocks inside the chosen sub-graph (they are split or merged) to further complicate analysis and finally each basic block updates the dispatcher's context to reflect the relative basic block relationships. The final resulting graph offers no clues about the structure of the algorithm, but has the same logic [34].

CFG flattening is frequently used, together with *opaque predicates*, to insert dead code paths in the CFG.

Code Transposition An obfuscator using the technique called *code transposition* effectively reorders the sequence of a program's original code instructions without changing its behaviour [33]. To achieve this two approaches are commonly followed:

- Randomly shuffling the instruction and then recovering the original execution order by inserting unconditional branches. This is easily defeated restoring the original program by removing (and following) the unconditional branches.
- Choosing and reordering independent instructions that have no impact on the others. This approach is harder to implement given the complexity of finding independent instructions, but it is more effective.

Code Virtualization *Code virtualization* consists in transforming a program's binary code (compiled for a specific machine) into a different binary representation that is understood by an interpreter embedded in the code. More specifically, the instruction set from the source machine is converted into a new, randomly chosen, instruction set. This means that a specific block of Intel x86 instructions, for example, can be converted into a different instruction set, preventing

an analyst/attacker from recognizing the generated virtual opcodes after the transformation from x86 instructions [34].

Usually, only some specific blocks of the program's code are virtualized (and not the whole program) and inserted back into the executable file alongside the associated interpreter. At run time, the interpreter assumes execution control and translates the virtualized code back to the original byte code before executing it.

When an analyst tries to decompile a virtualized block of code, however, he will not find the original x86 instructions. Instead, he will find a completely new instruction set he is not able to recognize even when using decompilers. This will force him to identify how each opcode is executed and how the specific interpreter works for each protected application.

Some examples of code virtualization tools include VMProtect and CodeVirtualizer.

Code Integration *Code integration*, one of the most sophisticated obfuscation techniques, was first introduced by *Win97/Zmist* malware. A malware using this technique first decompiles the target program into a set of manageable objects, it then inserts itself between them and finally reassembles the code [33].

2.2.9 Obfuscated Malware

The huge amount of malware released in the wild since the creation of the first virus in 1960s can be split into two generations. More specifically, the first generation malwares were static, their code and behaviour did not change. The more sophisticated second generation malwares, on the other hand, change their internal structure between one variant and the other maintaining the same malicious behaviour in order to avoid detection.

Encrypted Malwares

The first second-generation malwares ever existed exploited encryption in order to evade detection by signature-based antivirus scanners. An encrypted malware typically consists of two parts: the encrypted main body and a decryption code (also called *decryptor*). The *decryptor*'s objective is to recover the original malware code from the encrypted body whenever the infected file is run [31].

Moreover, to hide from signature-based scanners, encrypted malware encrypts its code using a different key at each infection, thus creating a unique encrypted body. The decryption routine (*decryptor*), however, remains the same from one generation to another. This means that encrypted malwares can be detected with signature-based scanners by searching for the decryptor's code pattern [22].

The first known malware to exploit encryption for detection evasion was CASCADE which spread in the 1980s and early 1990s.

Packed Malwares

Malware authors are nowadays increasingly exploiting packers (or even multiple packers at once) to produce numerous variants of the same original malware code [5]. As stated by Perdisci, et al. [35], more than 80% of the new malware currently discovered are actually packed versions of already existing malware.

As already metioned previously, packers are used to compress original executable files into smaller sizes. Moreover, encryption is sometimes also applied to the compressed version of the files in order to make the unpacking process more difficult.

While many malware authors frequently use commercial and readily available packers to generate new malware variants, it is not uncommon to see malware authors writing and using custom packers. This fact can be exploited by analysts to detect if a file is malicious, without further analysis, based on the fact that benign software vendors would almost never use custom packers.

Oligomorphic Malwares

Malware authors tried to overcome the short comings of encrypted malware developing malicious programs that can mutate the used decryptor from one variant to another. Initially the decryptor could only be changed slightly. However, a common method used by *oligomorphic malware*, also called '*semipolymorphic*', to provide more diverse decryptors is, in practice, to randomly select one decryption routine at infection time from a set of pre-defined decryptors [22].

However, this type of malware is able to generate at most few hundreds different decryptors. For example the virus called *Win95/Memorial* was capable of constructing up to 96 different decryptor patterns. This means that signature-based detection techniques are still able to detect *oligomorphic malwares* by generating the signature of all the decryptors utilised by the malware strain [31]. Still, signature based techniques are not an effective approach to detect *oligomorphic malware* [5].

Polymorphic Malwares

The *oligomorphic* malware limitations led malware authors to develop a more advanced type of malware called *polymorphic*, which, similarly to *oligomorphic* malwares, consist of two parts: the malware encrypted main body and the decryptor. The decryptor is again run when the malware is executed and it enables the execution of the original malware code decrypting the encrypted body. When replication occurs, the malware encrypts its code with a different key, generates the new associated decryptor and encloses it in the new malware variant code. However, in this type of malwares countless numbers (millions) of distinct decryptors can be generated by using a powerful toolkit called 'the *Mutation Engine* (*MtE*)'.In particular the mutation engine is responsible of preventing signature based detection by rearranging the decryptor code using different obfuscation methods including, for example, *dead-code insertion*, *register reassignment*, *subroutine reordering*, *instruction substitution*, *code transposition/integration*, etc. [31]. The malware appearance is thus changed at each infection [22].

Even though *polymorphic* malwares can create a large number of different decryptors effectively hindering signature matching techniques, still the constant malware body, which appears after decryption, can be used for detection. In particular, by using emulation techniques, it is possible to execute the malware in a '*Sandbox*' without resulting in any harm to the system. As soon as the constant malware body is decrypted and loaded into memory, the common detection techniques, such as signature based scanning, can be applied [5].

Various armouring techniques are thus used by malware authors to prevent detection by emulation, however most antivirus scanners are now capable of addressing also these techniques effectively defeating *polymorphic* malwares.

Metamorphic Malwares

After the *oligomorphic* and *polymorphic* malware types were effectively defeated, malware authors designed a new and more advanced approach: *metamorphic* malware. This, similarly to *polymorphism*, uses obfuscation techniques to create new variants of the original malware in order to evade detection [31]. However, in this case, instead of generating new decryptors, it is the malware body itself to be mutated through generations to appear different while having the same behaviour and functionality. *Metamorphic* malware is in fact said to be *body-polymorphic*. In practice the malware code logic is maintained while its appearance is changed using obfuscation techniques such as *dead-code insertion*, *register reassignment*, *code transposition* and more. This way, every generated malware variation appears different making signature based detection ineffective [22]. Moreover, many *metamorphic* malwares are also capable of interleaving their own code inside host programs, thus making detection even harder.

However, *metamorphic* malware, in order to efficiently evolve its code it needs to be able to recognise, parse and mutate its own body during propagation. This is far from being easy. Moreover, creating a true *metamorphic* malware without arbitrarily increasing its code size is also challenging [5].

Chapter 3

Detection Techniques

Malware detection is the process of identifying malicious from benign code with the purpose of protecting systems from malwares and/or eventually recover from their nefarious effects [5].

In order to counter malware attacks and threats, in recent years many anti-malware tools have been developed. Many of these are based on static features (such as signatures) with the assumption that most malware is static, as they don't mutate/change significantly at infection/replication time [22].

However, attackers are nowadays increasingly using the more sophisticated second generation malwares, which strongly mutate at each infection. Researchers and anti-malware software developers are thus focusing their attention on the creation of more advanced tools capable of detecting these types of evolving malwares.

3.1 Integrity Checker

When compromising a computer system or network some changes are inevitably made within the target environment. This implies that systems, like *integrity checkers*, that rely on actively monitoring changes made to existing files within the target operating system, can be used to perform intrusion detection [5].

Generally, *integrity checkers*, use hashing functions like the *md5* sum, *Sha1* or *Sha256* to calculate the digest of files and/or executables which are then stored in a database of digests. Program and file digests are then periodically re-calculated and compared against the ones in the database looking for modifications. If the digest of a file is different and no software updates nor patches were applied, then the file was probably tampered with.

Integrity checkers present a number of challenges:

- The system state in which the initial file digests are calculated has to be considered clean. However, this is difficult to be guaranteed.
- The application of system (and software) updates and patches, which modify system files and programs, must be followed by an update of the digests database, otherwise there will be a very high false positive rate.
- The digests database needs to be stored securely and there has to be an offline (and safe) backup, otherwise there would be a single point of failure.

Integrity checking can be considered as an important tool for detecting any system modifications, but it is more an incident recovery method rather than a malware intrusion/infection prevention method.

3.2 Signature-based Detection

Signature-based detection is the simplest and most widely used method in commercial anti-virus software (together with *heuristic-based* techniques) but is becoming less and less effective as the number of malware variants and second generation malwares increases [36].

Signature-based detection relies on *signatures* - specific unique byte code sequences/strings extracted from malware samples - to detect the presence of malicious files in a system. *Signatures* are typically created using static analysis techniques and are selected to be long enough to uniquely characterize a specific malware family with respect to benign programs (and other malware families). The *signatures*, which are crated by malware experts from a significant number of already identified malware samples, are saved in a *signature database* and deployed in anti-malware tools. These tools in turn scan the files in the target systems and consider as malicious any file that matches one of the known signatures [5]. This implies that the database of signatures must be maintained and frequently updated, especially whenever new malware variants are identified and new signatures are generated in order to detect them.

Some *signature-based* algorithms require an exact match between the signature of the analysed sample and one of the known signatures, others instead make use of *wildcard* characters to detect slight variations allowing them to detect even some second generation (evolving) malwares.

Anyway, the signature based detection approach is fast, easy to use and has a high positive rate; however, given that the number of known malwares is increasing so fast, it is quickly becoming time-consuming, expensive and impractical. Moreover, it is a completely reactive technique which is unable to counter threats/attacks from new malware families/variants until they cause damages. Additionally, most second-generation malwares are able to escape this type of detection [22].

3.2.1 Yara Rules

YARA is a widely accepted open-source *signature-based* malware analysis tool which has emerged in recent years thanks to its flexible and customisable nature. It allows malware analysts/researchers to develop malware "descriptions" based on text or binary patterns, commonly referred to as *Yara rules*. *Yara rules*, which combine simple regular expression matching with logic rules, can be used to identify specific malware families, the presence of *CVE*s, specific functionality signatures or even generic maliciousness indicators. Given the success obtained by this technique, many commercial malware analysis tools nowadays support *Yara rules* natively [37].

Yara rules can be generated either manually or automatically. Generating rules manually obviously requires high expertise, whereas generating them using automated tools is a relatively easy task. However, automatically generated rules are not guaranteed to be effective and may require post-processing operations for their optimization [38].

Malware analysts typically create *Yara rules* manually by reverse engineering malware samples looking for common *Indicator of Compromise (IoC)* strings. This is followed by the development and iterative refinement of the rules which are considered effective based on their coverage and false positive rate on a dataset of malicious, benign and out-of-family samples. Developing effective *Yara rules* can therefore be challenging and very time consuming, even for expert users with years of experience [39].

Yara Rules syntax

Listing 3.1 presents an example of the syntax of a simple *Yara rule*.

```
1  rule RuleName
2  {
3    meta:
4    description = "description of rule"
5    author = "name"
6    date = "dd/mm/yyyy"
7    reference = "url"
8
```

```
    strings:
    $text_string1 = "text1 you wish to find in malware"
    $text_string2 = "text2 you wish to find in malware"

    $hex_string1 = {hex1 you wish to find in malware}
    $hex_string2 = {hex2 you wish to find in malware}

    $reg_exp_string1 = /regular expression1 you wish to find in malware/
    $reg_exp_string2 = /regular expression2 you wish to find in malware/

    condition:
    $text_string1 or $text_string2 or
    $hex_string1 or $hex_string2 or
    $reg_exp_string1 or $reg_exp_string2
}
```

Listing 3.1: YARA Rules Syntax

As it can be seen in the above example, *Yara rules* must start with the keyword '*rule*', followed by the actual *RuleName*, which is the rule identifier. The *RuleName*s follow the same lexical conventions of the *C* programming language. They are, in fact, case sensitive, they cannot exceed 128 characters and they can contain only alphanumeric characters (with the addition of the underscore character), with the exception of the first character which cannot be a digit. Furthermore there is a list of *YARA* reserved keywords that cannot be used as identifiers [40].

Yara rules main body contains three sections: *meta*, *strings* and *condition*.

Meta section The rule author can include additional information about the rule as a list of attribute-value pairs - also called *metadata* - in the *meta* section, at the top of the rule. The values can be strings, integers or boolean values. The metadata, however, cannot be used in the condition section [41].

Some commonly used meta tags are, for example, "author" and "description", which convey information about the author and purpose of the rule. Moreover, malware analysts sometimes also leave tags with the hashes of the malicious files used for the creation of the rule, or references to blog posts with similar information [37].

Strings section This section contains the strings/patterns/signatures that a file must contain to 'trigger' the rule. This section is optional and can be omitted if it is not necessary. *YARA* supports searching for 3 string types: *Hexadecimal Strings*, *Text (ASCII) Strings* and *Regular Expressions*.

- *Hexadecimal Strings*: *Hexadecimal Strings* will match hexadecimal characters/sequences of raw bytes in the file being analysed. Example:

```
rule ExampleRule
{
  strings:
  $my_hex_string = { E2 34 A1 C8 23 FB }

  condition:
  $my_hex_string
}
```

Listing 3.2: YARA Hexadecimal example

Three special flexible formats, namely *wildcards*, *jumps* and *alternatives*, can be used to complement the search.

  - ***Wildcards*** are represented by the '**?**' symbol. They indicate that some bytes in the pattern are unknown and should match anything. For example:

```
rule WildcardExample
{
  strings:
  $hex_string = { E2 34 ?? C8 A? FB }

  condition:
  $hex_string
}
```

Listing 3.3: YARA Hexadecimal Wildcard example

– ***Jumps*** are used in circumstances when the values of the pattern are known but their length varies. For example:

```
rule JumpExample
{
  strings:
  $hex_string = { F4 23 [4-6] 62 B4 }

  condition:
  $hex_string
}
```

Listing 3.4: YARA Hexadecimal Jump example

In particular, in listing 3.4, the value '*[2-3]*' indicates that any arbitrary sequence from 2 bytes to 3 bytes long can occupy the sequence at that position.

– ***Alternatives***, whose syntax resembles regular expressions, are used in situations in which the author wants to provide different alternatives for a given fragment of the hex string. For example:

```
rule AlternativesExample
{
  strings:
  $hex_string = { F4 23 ( 62 B4 | 56 ) 45 }

  condition:
  $hex_string
}
```

Listing 3.5: YARA Hexadecimal Alternatives example

In particular, in listing 3.5, the value '*(62 B4 | 56)*' indicates that one sequence between '*62 B4*' and '*56*' can occupy that position.

- *Text Strings*: Text strings are generally readable sequences of ASCII characters which are then matched in the condition section [37].

  Example of a rule matching an ASCII-encoded, case-sensitive string:

```
  rule TextExample
{
  strings:
  $text_string = "foobar"

  condition:
  $text_string
}
```

Listing 3.6: YARA Text Strings example

Additionally, in order to specify how *YARA* should search for strings, some modifiers can be added at the end of each string definition. Moreover, even more than one modifier can be used in combination. Here are described some of the available modifiers:

– *nocase*: Text strings in YARA are, by default, case-sensitive. However it is possible to search for strings in case-insensitive mode by appending the modifier '*nocase*' at the end of the string definition, in the same line. Example:

```
rule CaseInsensitiveTextExample
{
  strings:
  $text_string = "foobar" nocase

  condition:
  $text_string
}
```

Listing 3.7: YARA nocase example

– *wide*: The '*wide*' modifier can be used to search for strings encoded with two bytes per character (also known as *wide character strings*), which are typically found in many executable binaries.

For example, if the string "Borland" appears in the file encoded as two bytes per character, then the following rule will match:

```
rule WideCharTextExample1
{
  strings:
  $wide_string = "Borland" wide

  condition:
  $wide_string
}
```

Listing 3.8: YARA Wide Character Strings example

– *xor*: *YARA* can also encode text before searching it in the analysed file. The '*xor*' modifier, for example, can be used to search for strings with a single byte XOR applied to them.

The following rule will search for every string resulting from a single-byte XOR applied to the string "This program cannot":

```
rule XorExample1
{
  strings:
  $xor_string = "This program cannot" xor

  condition:
  $xor_string
}
```

Listing 3.9: YARA XOR-ed Strings example

– *base64*: The '*base64*' modifier can be used to search for strings that have been base64 encoded.

For example, the following rule searches for all the possible base64 permutations of the string "This program cannot":

```
rule Base64Example1
{
  strings:
  $a = "This program cannot" base64

  condition:
  $a
}
```

Listing 3.10: YARA Base64 encoded Strings example

- *Regular Expressions*: Starting from version 2.0, YARA has been complemented with its own regular expression engine, which is one of its most powerful features. *Regular expressions* are defined in the same way as text strings, but enclosed in forward slashes instead of double-quotes [38].

  Example:

```
rule RegExpExample
{
  strings:
  $re1 = /md5: [0-9a-fA-F]{32}/
  $re2 = /state: (on|off)/

  condition:
  $re1 and $re2
}
```

Listing 3.11: YARA Regular Expression

Conditions section The last section of *YARA rules*, which is the only one really required, contains the rule conditions that determine when the rule gets triggered. These conditions are Boolean expressions similar to those used in programming languages [41]. Through the use of all the usual logical and relational operators, conditions can be made arbitrarily complex in order to accommodate the author specific needs [37].

Inside the *Conditions* section, among other things, it is possible to:

- **Count strings** Sometimes it is necessary to know how many times a string appears in the analysed file, not only if it is present or not. The number of occurrences of each string defined in the string section can be retrieved by using a variable whose name is the string identifier with a '#' character in place of the initial '$' character.

  For example:

```
rule CountExample
{
  strings:
  $a = "dummy1"
  $b = "dummy2"

  condition:
  #a == 6 and #b > 10
}
```

Listing 3.12: YARA Count example

- **Check String at specific offset/in offset range**: It is sometimes necessary to know if a particular string is available at some specific offset of the file or at some virtual address within the process address space. In such situations it is possible to use the '*at*' operator.

  The '*in*' operator, on the other hand, allows to search for a specific string within a range of offsets or addresses, rather than at an exact one.

  Examples:

  – The rule in listing 3.13 fires if the '*a*' string is located at offset 100 and '*b*' at offset 200 of the running process.

```
rule AtExample
{
  strings:
  $a = "dummy1"
  $b = "dummy2"

  condition:
  $a at 100 and $b at 200
}
```

Listing 3.13: YARA At example

  – The rule in listing 3.14 is triggered if the '*a*' and '*b*' strings are found in memory locations between *0* and *100* and between *100* and *filesize*, respectively, of the running process main memory.

```
rule InExample
{
  strings:
  $a = "dummy1"
  $b = "dummy2"

  condition:
  $a in (0..100) and $b in (100..filesize)
}

```

Listing 3.14: YARA In example

- **Check file size**: '*filesize*' is a special variable that can be used in rules conditions, which holds the size of the file being scanned in bytes.

  Example:

```
rule FileSizeExample
{
  condition:
  filesize > 200KB
}

```

Listing 3.15: YARA Filesize example

- **Check a set of strings**: When it is necessary to know if a file contains a certain number of strings from a given set, the '*of*' operator can be used.

  Example:

```
rule OfExample
{
  strings:
  $a = "dummy1"
  $b = "dummy2"
  $c = "dummy3"

  condition:
  2 of ($a, $b, $c)
}

```

Listing 3.16: YARA Of example

Additional modules *YARA*'s core functionality can be extended through the use of modules. Some modules like the *PE module* and the *Cuckoo module* are officially distributed with YARA, however additional ones can also be created. Here are mentioned some useful (in this document context) Yara modules:

- **YARA with PE** Starting with version 3.0, YARA can parse Portable Executable (PE) files [40]. For example, the rule in listing 3.17 will check for the string "abc", will parse the PE file and look for "*CreateProcess*" and "*httpsendrequest*" function names in the import sections '*Kernel32.dll*' and '*wininet.dll*', respectively.

```
Import "PE"

rule PE_Parse_Check
{
  strings:
    $string_pe="abc" nocase

  condition:
    pe.imports("Kernel32.dll", "CreateProcess") and
    pe,imports("wininet.dll", "httpsendrequest") and
    $string_pe
}

```

Listing 3.17: YARA with PE example

- **YARA with PEiD** *YARA* can also be integrated with *PEiD* to check what packer was used to compile the malicious/suspicious executable [40].

Yara Rules Advantages and Disadvantages

1. ***Advantages***: *Yara rules* offer several advantages over other malware analysis techniques. Here are some of the most notable ones [38]:
   - *Yara rules* allow malware analysts to write flexible and custom rules in an easy and efficient way.
   - *Yara rules* are an open standard which works on most of the major operating systems such as Windows, Linux and Mac OS.
   - *Yara rules* can be easily integrated into Python and C/C++ programming languages.
   - *Yara rules* can be used both for static and dynamic malware analysis.
   - Several automatic tools have been developed, and are readily available, to automatically generate *Yara rules* easily and efficiently.
   - There are various public repositories of *Yara rules* which offer readily available rules for malware analysis.
2. ***Limitations***: *Yara rules*, however, have also some limitations. Here are some of the most notable ones [38]:
   - *Yara rules* are commonly written based on *IoC* (*Indicator of Compromise*) strings, however, malware authors can easily obfuscate, replace or encrypt these *IoC* strings in their creations in order to evade detection. This could make these rules less effective.
   - *IoC* strings are usually extracted from existing malware samples/families through the use of reverse engineering techniques. The use of these techniques in manually creating effective rules, however, requires a highly specialised skill-set and years of experience.
   - The effectiveness of *Yara rules* is generally influenced by the types and number of *IoC* strings included in the rules. However, achieving the right balance of both is a challenging task.
   - *Yara rules* are effective in detecting malwares which match known malware signatures. It may, however, completely miss new and unique malware variants.

Yara Rules Automatic Generators

There are various automatic *Yara rules* generator tools available. In the following the most notable ones will be briefly described:

***YarGen* Tool** *YarGen python-based* tool exploits some smart techniques, namely fuzzy regular expressions, Naive Bayes classifier and Gibberish Detector, to generate *Yara rules*.

The produced rules include features (strings and opcodes) common to malware samples that don't match with the provided goodware databases. A predefined number of features (generally up to 20 strings) are selected, based on their potential utility and a number of heuristics, to be combined and used by the rule in order to maintain a reasonable operation speed.

This tool is able to generate two types of rules: *basic rules* and *super rules*. *Basic rules* can generally target specific malware samples, where *super rules* are able to target a set of malware samples or a whole malware family [38].

The *yarGen* authors encourage its use as a starting point for rule construction, followed by manual adjustments to refine *yarGen*'s output [39].

1. *YarGen* tool *advantages*:

- It allows generation of *Yara rules* based on both opcodes and strings.
- It supports the use of PE (portable executable) modules, which are used to interpret Windows operating system executables such as *DLL* and *COM* files.
- It can be integrated with other anti-malware software in order to improve its effectiveness.
- It reduces the false positive rate by checking all strings against databases of goodware samples.
- It is deployed as a simple and easy-to-use python script that can be run through a command-line interface.

2. *YarGen* tool *disadvantages*:

- It requires post-processing of the generated rules for increasing their effectiveness.
- It requires significant resources for generating opcode-based rules and for loading goodware files.
- The rule generation process is slow.
- The creation of super rules may cause redundancy and duplication of rules.
- All dependencies and built-in databases have to be installed in order for the tool to work successfully.

***YaraGenerator* Tool** This *python-based* tool uses string prioritization logic and code refactoring to generate *Yara rules* with a completely different signature for different file types, such as *EXE*s, *PDF*s, and *Email*s.

The generated *Yara rules* contain only strings (opcodes are not supported) extracted from malware samples that do not match with the provided database of strings from blacklisted files. In particular 30,000 blacklisted strings are contained in such database, arranged based on the different file formats. The produced *Yara rules* contain a large number of strings which are selected randomly. In fact, no score computation takes place in order to weight the different strings [38].

1. *YaraGenerator* tool *advantages* :

- It can generate specialised rules for specific file formats.
- It supports the use of PE (portable executable) modules, which are used to interpret Windows operating system executables such as *DLL* and *COM* files.
- It reduces the false positive rate by checking all strings against databases of blacklisted strings.
- It is deployed as a simple and easy-to-use python script that can be run through a command-line interface.

2. *YaraGenerator* tool *disadvantages*:

- It requires post-processing of the generated rules for increasing their effectiveness.
- It generates rules based on a random selection of features (strings). This implies that the most appropriate strings may not be selected in many cases, thus making the produced rules less effective on average.
- It does not support the use of opcodes.
- It was developed as a work-in-progress project and has not been updated since.

***Yabin* Tool** This is another *python-based* tool, developed by the *Alien Vault Open Threat Exchange* (*OTX*) community, for the automatic generation of *Yara rules*.

In this case *Yara rules* are created by finding rare functions in specific malware samples or families. Functions are recognised by checking specific bytes sequences called *function prologues*, which define the start of the code of a function. For example, the byte sequence '*55 8B EC*' usually specifies the start of a function in programs compiled by Microsoft Visual Studio.

The generated *Yara rules* include those strings common to malware samples that don't match with the provided whitelist of commonly used library functions. Such whitelist was obtained from 100 GB of non-malicious software in order to exclude common library functions. The produced *Yara rules* contain a list of hexadecimal strings to be compared against suspicious files looking for similarities in their byte-sequences [38].

1. *Yabin* tool *advantages*:
    - It can be used to cluster malware samples based on the reuse of their code.
    - The list of patterns to search for can be extended during the rule post-processing phase.
    - With the purpose of excluding commonly used library functions in the produced rules, a large whitelist obtained from numerous non-malicious executable files is provided with the tool.
    - It is deployed as a simple and easy-to-use python script that can be run through a command-line interface.
2. *Yabin* tool *disadvantages*:
    - It requires post-processing of the generated rules for increasing their effectiveness.
    - Some specific file types/formats may not be supported.
    - The created rules contain only function prologues. No other string types are used.
    - Since it relies on function prologues, it works only with unpacked executables.
    - It is not designed to work on *.NET* executables, *Java* files and *Microsoft* documents.
    - It was mainly developed for research and testing purpose, not for production use.

***AutoYara* Tool** Compared to the previously mentioned tools like *YarGen*, which rely on a number of heuristics and string features, *AutoYara* tool makes larger rules using the redundancy and conjunction of components to achieve extremely low false-positive rates [39].

The two primary concerns of the *AutoYara* authors while designing this tool were:

1. Yara rules that generate a lot of false positives could slow down the investigation
2. Malware analysts often have few samples (≤ 10) when creating a Yara rule

AutoYara authors thus developed a workflow composed of two steps: the first step leveraged recent works in finding frequent larger n-grams, for $n \leq 1024$, to find several candidate byte strings that could become features. In the second step a bi-clustering method, which consists of simultaneously clustering the rows and columns of an input data matrix, is used on those strings to construct the output rules. Most bi-clustering algorithms require the specific number of bi-clusters to be known in advance, and enforce no overlaps between bi-clusters. The *AutoYara* authors exploited an already existing bi-clustering algorithm extending it to work when the number of bi-clusters is not known *a priori* (the number of bi-clusters gets determined automatically) and to allow overlapping bi-clusters, discarding rows and columns that do not fit in any bi-cluster [39].

AutoYara uses *bi-clustering* because it allows to easily produce complex and effective logic rules that enable the creation of signatures with low false positive rates.

To build a good Yara rule, in fact, one needs to know:

1. which features should be used at all
2. which features should be combined into '*and*' statements (which reduce the False Positive Rate), and which should be placed into '*or*' statements (which increase the True Positive Rate)

Bi-clustering provides a simple approach to do this jointly over the features, rather than considering the features one at a time. In particular, the features within a bi-cluster are combined into an '*and*' statement since they co-occur; moreover the '*and*' statements from multiple bi-clusters are placed into an '*or*' statement resulting in a "disjunction of conjunctions" rule formulation.

1. *AutoYara* tool *advantages*:
   - It is fast, allowing it to be deployed even on low-resource equipment (like remote networks).
   - It was designed with the intent of producing *Yara rules* with low false positive rates.
   - It was designed to be able to generate *Yara rules* from as few as ≤ 10 available samples.
2. *AutoYara* tool *disadvantages*:
   - It requires post-processing of the generated rules for increasing their effectiveness.
   - It a very recent tool, mainly developed for research purposes and not for production use.

3.3 Semantic Based Detection

Semantic-based malware detection aims at identifying malware by deducing the analysed code logic and comparing it to a database of already known malicious logic patterns. This technique, differently from signature-based detection which looks at the code syntactic properties, tracks the semantics of the program code instructions. This implies that *semantic-based* detection approaches are capable of overcoming obfuscation attempts and are even able of detecting unknown malware variants [5].

3.4 Behavioural Based Detection

Behavioural-based malware detection consists in the use of behavioural patterns for the identification of malicious software. This is done by dynamically analysing malware samples and extracting specific system/application behaviours and activities in order to form a '*behavioural signature*' of a malware strain. New samples are then analysed in the same way and classified as malicious if their behavioural pattern is similar to the *behavioural signature* of a known malware [5].

Behavioural-based detection is for the most part immune to obfuscation attempts. However, it's applicability is limited since it is based on the time consuming dynamic analysis and on the challenging task of determining the unsafe activities/behaviours to consider within the target environment.

3.5 Heuristics-based Detection

As opposed to traditional *signature-base* detection methods which identify malware by looking in the code for specific bytes/strings, *heuristic-based* detection uses rules and/or algorithms to search for commands or instructions not commonly found in harmless applications, thus indicating possible malicious intents [42].

Heuristic-based anti-malware tools may exploit different scanning techniques such as:

- *File analysis (static heuristic analysis)*: the suspicious program is disassembled and its source program is examined looking for known malware patterns (stored in a heuristic database). If the percentage of matched code exceeds a predefined threshold then the code is marked as probably infected [43].
- *File emulation (dynamic heuristic analysis)*: in this approach, the suspicious piece of code is examined in a virtual machine (or sandbox) looking for suspicious operations such as attempts at executing other executables, at changing the Master Boot Record, at concealing themselves etc. that are uncommon in benign programs.
- *Genetic signature detection*: this technique is designed to spot different malware variations within the same family using previous malware definitions [44].

Heuristic analysis is a more effective technique than the signature-based approach for the detection of unknown malware, particularly for encrypted and polymorphic variants [22]. Nowadays it can be found in most mainstream antivirus solutions in the market, combined with signature-based scanners in order to improve detection rate while reducing false alarms [5].

3.6 Machine Learning

In recent years, the rapid proliferation and increased sophistication of malicious software, coupled with the rising popularity of machine learning techniques in many fields, led to the adoption of more general ML-based approaches to malware detection in addition to the use of manually generated signatures and heuristics [3].

In particular, the application of machine learning for Information Security (ML-Sec) methods to perform malware detection generally consists in training a highly parametrized ML classifier to reliably (as much as possible) predict a binary label (malicious or benign) using features extracted from sample files. In order to do this the classifier parameters are numerically optimized to learn general concepts of *malware* and *benignware*, by minimizing the misclassification loss between predictions and the actual ground truths. This is based on the assumption that, if the samples are well labelled and malware/benignware samples in the training set are similar enough to those seen at test/deployment time, the learned detection function should work well on unseen samples [2].

Most static ML-Sec classifiers work on learned embeddings over portions of files (e.g. headers), learned embeddings over full files, or most commonly, on pre-engineered numerical *feature vectors* designed to summarize the content from each file. Learned embeddings generally are the result of convolutional architectures which do not presume a fixed file structure. However, the process of embedding features directly from inputs is expensive, and does not scale gracefully. Moreover, generic bytes do not present structural localities/hierarchies typical of images and text inputs that can be exploited by convolutional filters. Pre-engineered feature vector representations, on the other hand, quickly distil content useful for classification from each file [45]. There are various possible ways to statically craft feature vectors, for example:

- tracking per-byte statistics over sliding windows
- using byte histograms and/or n-gram histograms
- treating bytes as pixel values in an image
- computing opcode and function call graph statistics
- computing symbol statistics
- extracting hashed/numerical metadata values
- extracting hashes of delimited tokens
- etc.

The process of transforming a sample file into its numerical feature representation is called *feature extraction* and consists of some numerical transformations that preserve the aggregate and fine-grained information of each sample [2].

ML methods generally require many high quality samples in order to train effective models. When creating datasets for these models, labels are often collected from vendor aggregation feeds, which combine detection results from various vendors for each malware sample. This can be done, for example, by using a $1-/5+$ criterion or by using statistical estimation methods. The $1-/5+$ criterion works as follows: if a file has one or fewer vendors reporting it as malicious, the file is labelled as 'benign'; on the other hand, if a sample has five or more vendors reporting it as malicious, the file is labelled as 'malicious'. Moreover, it is common practice to introduce a time lag to let vendors update their models to account for new malware samples. When deployed, classifiers are periodically re-trained on new data/labels to reflect the current malware trends [45].

The advantage of machine learning techniques with respect to signature engines, where the aim is to reactively blacklist/whitelist samples that hard-match manually-defined patterns (signatures), is that, by being more general, they are able to detect not only known malwares but also novel malware strains/variants, providing some degree of proactive detection [36].

Commercial anti-malware solutions/engines have nowadays integrated ML on top of standard detection methods (without replacing them) with the aim of enhancing the detection results, especially for second generation malwares and novel malware strains. Popular ML techniques employed by such tools are, for example, deep neural networks (DNN), boosted decision tree ensembles, Naïve Bayes models, Data Mining approaches and Hidden Markov Models. Moreover, multiple vendors in the IT security industry nowadays have dedicated ML-Sec teams [22].

The following recent static ML-based malware detection methods will be described in the next sections:

- ALOHA: Auxiliary Loss Optimization for Hypothesis Augmentation
- Automatic Malware Description via Attribute Tagging and Similarity Embedding
- Learning from Context: Exploiting and Interpreting File Path Information for Better Malware Detection

3.6.1 ALOHA: Auxiliary Loss Optimization for Hypothesis Augmentation

In a recent work called **ALOHA: Auxiliary Loss Optimization for Hypothesis Augmentation**, [2], Rudd et al. observed that, although ML-based malware detection is frequently framed as a binary classification task (using a simple binary cross-entropy loss function), there are often a number of other sources of contextual metadata for each input sample available at training time, beyond just aggregate malicious/benign labels. Such metadata might include malicious/benign labels from multiple sources (e.g. from various security vendors), malware family information, temporal information, counts of affected endpoints, and associated tags. However, this metadata is, in many cases, not available at deployment time, making it difficult to include it as input features. The authors thus proposed exploiting the metadata collected from threat intelligence feeds as auxiliary targets in a multi-target learning approach. Simultaneously optimizing classifier parameters for multiple targets (labels) while training a model may, in fact, have a regularizing effect leading to better generalization, particularly if the auxiliary targets are related to the main target of interest.

In practice, in their work [2] the authors fit a deep neural network with multiple additional targets derived from metadata in a threat intelligence feed for Portable Executable (PE) malware and benignware. The additional losses include a multi-source malicious/benign loss, a count loss on multi-source detections, and a semantic malware attribute tag loss.

Inner Workings

The model presented in [2] (fig. 3.1) is composed of a base feed-forward neural network consisting of 5 blocks, each composed of Dropout, a dense layer, batch normalization, and an exponential linear unit (ELU) activation, with 1024, 768, 512, 512, and 512 hidden units respectively. This base topology applies the function $f(\cdot)$ to the input feature vector $\mathbf{x}$ (of size 1024 in this case) to produce an intermediate 512 dimensional representation of the input file $\mathbf{h} = f(\mathbf{x})$. An additional block for each output of the model, consisting of one or more dense layers and activation functions, is then appended on top of the base net. This composition of the base topology and the target-specific 'heads' is denoted as $f_{target}(\mathbf{x})$.

The output for the main malware/benign prediction task - $f_{mal}(\mathbf{x})$ - is always present (it constitutes the baseline model) and consists of a single dense layer followed by a sigmoid activation function on top of the base shared network. On top of that one or more auxiliary outputs are added with similar structure as described above: one fully connected layer (two for the *tag* prediction task) with a task-specific activation function. Finally, multi-task losses are produced by computing the sum, across all tasks, of the per-task loss multiplied by a task-specific weight (1.0 for the malware/benign task and 0.1 for all other tasks) [2].

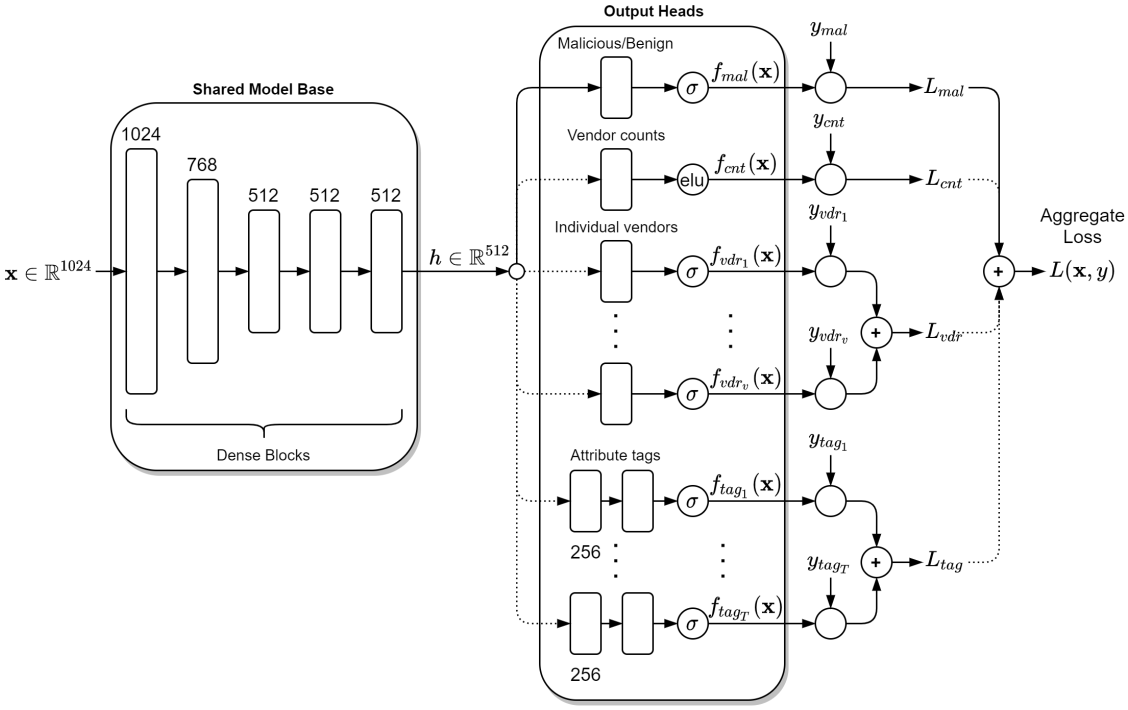


Figure 3.1: ALOHA model architecture

Malware Loss The task of predicting if a given binary file, represented by its features $\mathbf{x}^{(i)}$, is malicious or benign is optimized by minimizing the binary cross-entropy loss between the malicious/benign output label of the network $\hat{y}^{(i)} = f_{mal}(\mathbf{x}^{(i)})$ and the malicious ground truth label $y^{(i)}$. This results in the following loss for a dataset with M samples:

$$\begin{aligned} L_{mal}(X, Y) &= \frac{1}{M} \sum_{i=1}^{M} l_{mal}(f_{mal}(\mathbf{x}^{(i)}), y^{(i)}) \\ &= -\frac{1}{M} \sum_{i=1}^{M} y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}). \end{aligned} \tag{3.1}$$

A "1-/5+" criterion was used for labelling a given file as malicious or benign [2].

Vendor Count Loss The first additional target the authors of [2] used is the count of 'malicious' reports for a given sample from the vendor aggregation service. This is based on the assumption that the more a sample gets reported by vendors, the more likely it is to be malicious. To model count data, the authors used the Poisson noise model parametrized by the parameter μ, where μ is the mean and variance of the Poisson distribution. The probability of an observation of y counts is therefore:

$$P(y|\mu) = \mu^y e^{-\mu}/y!. \tag{3.2}$$

The model is then trained to *estimate* μ for each sample $\mathbf{x}^{(i)}$ such that the likelihood of $y^{(i)}|\mu^{(i)}$ is maximized (or, equivalently, the negative log-likelihood is minimized). The output of the model vendor count head, for sample i, is denoted as $f_{cnt}(\mathbf{x}^{(i)})$. Thereafter, an activation function $a(\cdot)$, which maps $f_{cnt}(\mathbf{x}^{(i)})$ to the non-negative real numbers, is used, so the final approximation for parameter μ is: $\mu^{(i)} = a(f_{cnt}(\mathbf{x}^{(i)}))$. In particular, the activation function used in [2] is an exponential linear unit activation (ELU).

Letting $y^{(i)}$ denote the actual number of vendors that deemed sample $\mathbf{x}^{(i)}$ malicious, the corresponding negative log-likelihood loss over the dataset is:

$$\begin{aligned} L_p(X, Y) &= \frac{1}{M}\sum_{i=1}^{M} l_p(a(f_{cnt}(\mathbf{x}^{(i)})), y^{(i)}) \\ &= \frac{1}{M}\sum_{i=1}^{M} \mu^{(i)} - y^{(i)}\log(\mu^{(i)}) + \log(y^{(i)}!), \end{aligned} \tag{3.3}$$

which is referred to in [2] as the *Poisson* or *vendor count* loss. In practice, the term $\log(y^{(i)}!)$ is ignored when minimizing this function loss since it does not depend on the network parameters.

Per-Vendor Malware Loss The authors of [2] identified a subset $\mathcal{V} = \{v_1, ..., v_V\}$ of 9 vendors that each produced a result for (nearly) every sample in their data. Each vendor result was added as additional target by adding an extra fully connected layer per vendor followed by a sigmoid activation function to the end of the shared baseline architecture. A per-vendor binary cross-entropy loss was then employed during training. The aggregate *vendors* loss L_{vdr} for the $V = 9$ selected vendors is simply the sum of the individual vendor losses:

$$\begin{aligned} L_{vdr}(X, Y) &= \frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{V} l_{vdr}(f_{vdr_j}(\mathbf{x}^{(i)}), y_{v_j}^{(i)}) \\ &= -\frac{1}{M}\sum_{i=1}^{M}\sum_{j=1}^{V} y_{v_j}^{(i)}\log(\hat{y}_{v_j}^{(i)}) + (1 - y_{v_j}^{(i)})\log(1 - \hat{y}_{v_j}^{(i)}), \end{aligned} \tag{3.4}$$

where l_{vdr} is the per-sample binary cross-entropy function and $f_{vdr_j}(\mathbf{x}^{(i)}) = \hat{y}_{v_j}^{(i)}$ is the output of the network that is trained to predict the label $y_{v_j}^{(i)}$ assigned by vendor j to input sample $\mathbf{x}^{(i)}$ [2].

Malicious Tags Loss Finally, further additional targets were provided in the form of malicious tags. This was done in an attempt to exploit information contained in family detection names provided by different vendors. In particular, the tags used as auxiliary targets in [2] are: *flooder*, *downloader*, *dropper*, *ransomware*, *crypto-miner*, *worm*, *adware*, *spyware*, *packed*, *file-infector* and *installer*.

These tags were created by parsing individual vendor detection names, using a set of 10 vendors which provided high quality detection names. After having extracted the most common tokens, the tokens not related to well-known malware family names were filtered out. Finally, a mapping between tokens and tags was created based on the authors experience.

The tag prediction task was then defined as a multi-label binary classification, since zero or more tags from the set of possible tags $\mathcal{T} = \{t_1, ..., t_T\}$ can be present at the same time for a given sample. This was implemented as a *multi-headed* architecture: two additional layers per

tag were added to the end of the shared baseline architecture, a fully connected layer of size 512-to-256, followed by a fully connected layer of size 256-to-1, followed by a sigmoid activation function. Each of the possible $T = 11$ tags had its own binary cross-entropy loss term. Finally, the aggregate tag loss was computed as the sum of the individual tag losses [2]. For the dataset with M samples the loss thus was:

$$\begin{aligned} L_{tag}(X, Y) &= \frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{T} l_{tag}(f_{tag_j}(\mathbf{x}^{(i)}), y_{t_j}^{(i)}) \\ &= -\frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{T} y_{t_j}^{(i)} \log(\hat{y}_{t_j}^{(i)}) + (1 - y_{t_j}^{(i)}) \log(1 - \hat{y}_{t_j}^{(i)}), \end{aligned} \tag{3.5}$$

where $y_{t_j}^{(i)}$ indicates if sample i is annotated with tag j, and $\hat{y}_{t_j}^{(i)} = f_{tag_j}(\mathbf{x}^{(i)})$ is the prediction of the network for that value.

Final Loss Finally, in [2] each model used a loss weight of 1.0 on the aggregate malicious/benign loss and 0.1 on each auxiliary loss. Therefore, when adding K targets to the main loss, the final loss that gets back-propagated through the model was:

$$L(X, Y) = L_{mal}(X, Y) + 0.1 \sum_{k=1}^{K} L_k(X, Y). \tag{3.6}$$

3.6.2 Automatic Malware Description via Attribute Tagging and Similarity Embedding

As explained by Ducau et al. in [3], in order to counter and remediate a malware infection on a system/network it is of vital importance to understand the nature (malware family and variant) of the attack in progress. In fact, knowing the malicious capabilities associated with each suspicious file found on a system/network gives important clues to the system end user, administrator or security operator that help define a remediation procedure, identify possible root causes, and evaluate the severity and potential consequences of the attack.

Machine learning detection methods, which have the potential to identify even new malware samples/variants as malicious, however, generally produce a simple binary malicious/benign label with no further information about the type of threat posed by malicious samples, which in turn does not allow the identification of relationships between malware samples. On the other hand, most commercial anti-malware solutions provide, when they alert about potentially harmful files detected in a machine, *detection names* coming from specific hand-written signatures created by reverse engineers to identify particular threats, that are theoretically useful for categorizing known malware variants. However, even these detection names are problematic. In fact, the increasing number of feature-rich malware and the fact that different vendors use differing malware naming conventions led to inconsistent and highly vendor-specific detection names. Moreover, it is now common to see detection names which only act as unique identifiers without providing actionable information about the type of potential harm the malicious sample is capable of doing.

The authors of [3] therefore proposed the use of SMART (Semantic Malware Attribute Relevance Tagging) tags, which are human interpretable, high level descriptions of the capabilities of a given malware sample, to approach malicious software description. These SMART tags, which were derived by leveraging the underlying knowledge encoded in detection names from different anti-malware vendors in the industry, are non-exclusive, meaning that one malware family can be associated with multiple tags and a given tag can be associated with multiple malware families. In [3], the authors defined a set of malicious tags $\mathcal{T}$, with $|\mathcal{T}| = 11$ different tags (or descriptive dimensions) of interest that they then used to describe malicious PE files: *adware*, *crypto-miner*, *downloader*, *dropper*, *file-infector*, *flooder*, *installer*, *packed*, *ransomware*, *spyware* and *worm*.

These tags were then used to train a Joint Embedding neural network to learn a low dimensional Euclidean representation (embedding) space in which malware samples with similar

characteristics are close to each other, having access only to the files' static binary representations; this in turn is used at test/deployment time for automatically predicting tags for new (unseen) files in real time. The representation of a malware sample in the embedding space can therefore be taught also as an implicit 'signature' describing its capabilities.

Multi-Label Classification The authors of [3] applied multi-label classification in order to perform semantic attribute tagging. The most trivial way of implementing a multi-label classification model is by learning one classifier per-label. This approach, however, is far from being efficient since the single per-tag classifiers are independently optimized. In this scenario, it is more common to use a single classifier with multiple outputs (multi-label learning) and multiple target losses which are combined and jointly optimized during training (multi-objective loss). This approach yields a more compact representation while also improving classification performance with respect to using independent classifiers. In [3], the authors used a multi-label deep neural network as baseline architecture.

An alternative approach to multi-label classification is to learn a compact shared vector (embedding) space representation on which to map both input samples and labels - a joint embedding - where similar content across modalities are projected into similar vectors in the same low dimensional space [3]. Then, at test/deployment time, in order to determine likely labels, a similarity comparison between vectors belonging to this learned latent space is performed, e.g. via inner product. In [3], the authors used a joint embedding model that maps malware tags and executable files into the same low dimensional Euclidean joint-embedding space for the malware description problem.

Tag Distillation from Detection Names Ducau et al. [3] relied on semi-automatic strategies, even if they are noisier than manual labelling, because they allowed them to label millions of files that can be then used to train a classifier. In particular, they designed a labelling function which annotates PE files using the previously mentioned set of tags $\mathcal{T}$ by combining information extracted from the detection names got from ten reputable anti-malware vendors.

The labelling process consisted of the three main stages presented in figure 3.2:

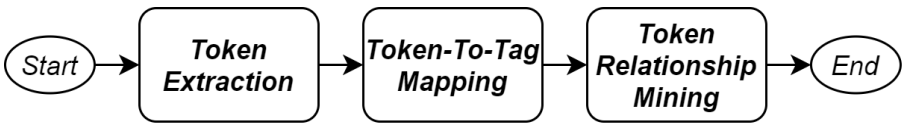


Figure 3.2: Tag Distillation from Detection Names Process

The token extraction phase consisted of normalizing and parsing the multiple detection names and converting them in sets of sub-strings. In a similar way to what the malware labelling tool AVClass, which was proposed by Marcos et al. in [46], does, the token-to-tag mapping stage uses rules created from expert knowledge by a group of malware analysts, that associate relevant tokens with the set of tags of interest. Finally, this mapping is extended by mining statistical relationships between tokens to improve tagging stability and coverage.

Tags Prediction - Inner Workings In [3], in order to predict, in a multi-label classification manner, zero or more tags per sample from the set of T possible tags $\mathcal{T} = \{t_1, t_2, ..., t_T\}$, the authors proposed two different neural network architectures, represented in 3.3 and 3.4, which they referred to as *Multi-Head* and *Joint Embedding* nets.

In particular, the *Multi-Head* (fig. 3.3) consisted of a base topology common to the prediction of all tags, and one output head per tag. The shared base topology, which can be thought of a feature extraction (or embedding) network that transforms the input features $\mathbf{x}$ into a low dimensional hidden vector $\mathbf{h}$, consisted of an input feed-forward layer of output size 2048, followed by a batch normalization layer, an ELU non-linearity and three blocks, each composed by dropout, a linear layer, batch normalization and ELU of output sizes 512, 128, and 32 respectively. Each

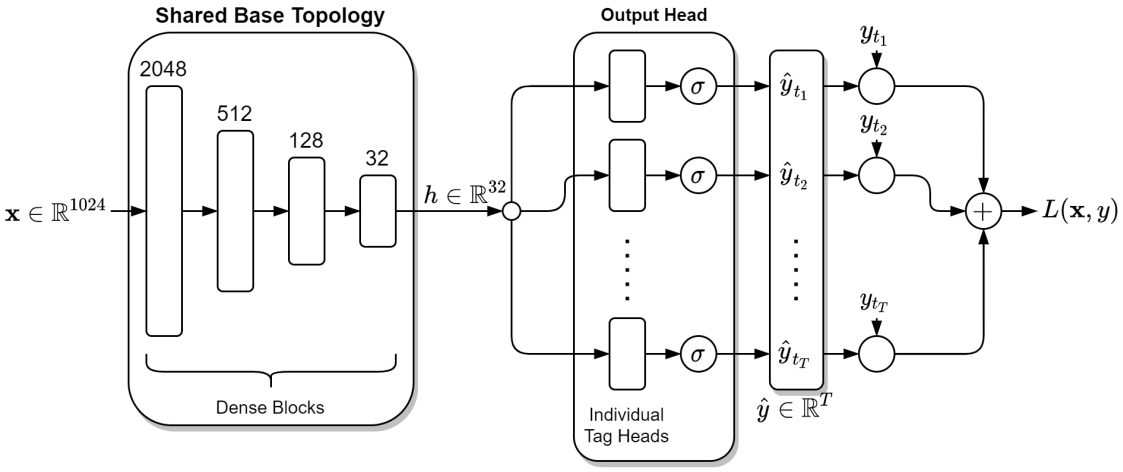


Figure 3.3: Multi Head model architecture

head, on the other hand, is a binary classifier that predicts the presence or absence of each tag and it consists of a linear layer (the same for each head) composed of the same type of basic blocks as in the main base architecture, but with output size 11 (the number of tags being predicted) and a sigmoid non-linearity instead of the ELU to compute the predicted probability for each label. The binary cross-entropy losses between each head output and the corresponding ground truth tags are computed and then added together to form the final loss.

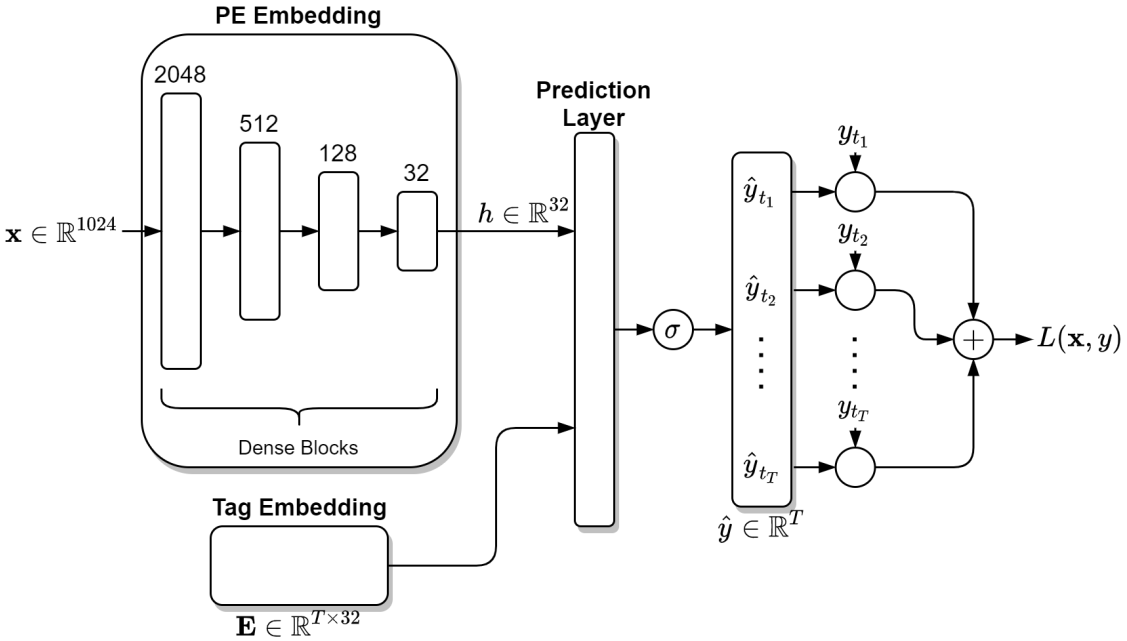


Figure 3.4: Joint Embedding model architecture

The *Joint Embedding* model (fig. 3.4), instead, was designed to map both the labels (malware tags) and the binary file features $\mathbf{x}$ to vectors in a joint Euclidean latent-space in such a way that, for a given similarity function, the transformations of semantically similar labels are close to each other, and the embedding of a binary file should be close to that of its associated labels in the same space. In practice, the *Joint Embedding* model consisted on a PE *embedding* network, a *tag embedding* matrix $\mathbf{E}$, and a *prediction layer*.

The PE embedding network, using the same base topology as the Multi-Head model, learns a non-linear function $\phi_\theta(\cdot)$, with parameters θ, that maps the input binary representation of the PE

executable file $\mathbf{x} \in \mathbb{R}^d$ into a vector $\mathbf{h} \in \mathbb{R}^D$ in low dimensional Euclidean space (with $D = 32$),

$$\phi_\theta(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}^D. \tag{3.7}$$

The tag embedding matrix $\mathbf{E} \in \mathbb{R}^{T \times D}$ of learnable parameters learns a mapping from a tag $t_n \in \mathcal{T} = \{t_1, ..., t_T\}$, to a distributed representation $\mathbf{e} \in \mathbb{R}^D$ in the joint embedding space (with $D = 32$).

$$\phi_E(t) : \{t_1, ..., t_T\} \to \mathbb{R}^D. \tag{3.8}$$

In practice, the embedding vector for the tag t_n is simply the n-th row of the tag embedding matrix, i.e. $\phi_E(t_n) = \mathbf{E}_n$.

Finally, the prediction layer compares both tag and sample embeddings ($\mathbf{e}$ and $\mathbf{h}$ respectively) and produces a similarity score. This is later run through a sigmoid non-linearity to estimate the probability that sample $\mathbf{x}$ is associated with tag t for each $t \in \mathcal{T}$. In the final model implementation presented in [3], the similarity score used was the dot product between the embedding vectors. The output of the network $f_n(\mathbf{x}|\theta, \mathbf{E})$ then was,

$$\begin{aligned} \hat{y}_n = f_n(\mathbf{x}|\theta, \mathbf{E}) &= \sigma(\langle \phi_E(n), \phi_\theta(\mathbf{x}) \rangle) \\ &= \sigma(\langle \mathbf{E}_n, \mathbf{h} \rangle), \end{aligned} \tag{3.9}$$

where σ is the sigmoid activation function, and $\hat{y}_n$ is the probability estimated by the model of tag t_n being a descriptor for $\mathbf{x}$.

Furthermore, the authors of [3] constrained the embedding vectors for the tags such that:

$$||\mathbf{E}_n||_2 \leq C, n = 1, ..., T, \tag{3.10}$$

with C equal to 1, which has a regularizing effect.

During training, the parameters of both embedding functions $\phi_\theta(\cdot)$ and $\phi_E(\cdot)$ are jointly optimized to minimize the binary cross-entropy loss for the prediction of each tag via back-propagation and stochastic gradient descent. The loss function to minimize for a mini-batch of M samples is:

$$\begin{aligned} L &= -\frac{1}{M} \sum_{i=1}^{M} \sum_{n=1}^{T} f_n(\mathbf{x}^{(i)}|\theta, \mathbf{E}) \log(t_n^{(i)}) + (1 - f_n(\mathbf{x}^{(i)}|\theta, \mathbf{E})) \log(1 - t_n^{(i)}) \\ &= -\frac{1}{M} \sum_{i=1}^{M} \sum_{n=1}^{T} \hat{y}_n^{(i)} \log(t_n^{(i)}) + (1 - \hat{y}_n^{(i)}) \log(1 - t_n^{(i)}) \end{aligned} \tag{3.11}$$

where $t_n^{(i)} = 1$ if sample i is labelled with tag t_n or zero otherwise, and $\hat{y}_n^{(i)}$ is the probability predicted by the network of that tag being associated with the i-th sample.

Finally, at test/deployment time, in order to get a vector of tag similarities for a given sample $\mathbf{x}$ with PE embedding vector $\mathbf{h}$, the matrix of tag embeddings $\mathbf{E} \in \mathbb{R}^{T \times D}$ is multiplied (dot product) by $\mathbf{h} \in \mathbb{R}^D$; the output is then scaled to obtain a prediction vector $\hat{\mathbf{y}} = \sigma(\mathbf{E} \cdot \mathbf{h}) \in \mathbb{R}^T$, where σ is the element-wise sigmoid function for transforming the similarity values into a valid probability. Each element in $\hat{\mathbf{y}}$ is then the predicted probability for each tag.

Evaluation of Tagging Algorithms The performance evaluation of tagging algorithms can be done along two orthogonal dimensions: *per-tag* or *per-sample*. The former aims at quantifying the tagging algorithm performance at identifying each tag, while the latter focuses on evaluating the tagging algorithm performance for a given sample, across all tags.

One suitable way to evaluate the *per-tag* performance of a model is by measuring the *Area Under the Receiver Operating Characteristic* curve (*AUC-ROC*, or simply *AUC*) for each of the tags being predicted. A *ROC* curve is created by plotting the *True Positive Rate* (*TPR*) against the *False Positive Rate* (*FPR*). Furthermore, given the binary True/False nature of the target value for the n-th tag of a given sample ($t_n \in \{0, 1\}$), the typical binary classification evaluation

metrics such as 'Accuracy' (eq. 3.12), 'Precision' (eq. 3.13), 'Recall' (eq. 3.14), and 'F-score' (eq. 3.15) can also be computed. However, in order to compute these metrics, the output probability prediction needs to be binarized first. This can be done by simply choosing a threshold, independently for each tag, so that the *FPR* on the validation set is, for example, 0.01 and then using the resulting 0/1 predictions. The equations of the mentioned binary cross-entropy statistics are:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{3.12}$$

$$precision = \frac{TP}{TP + FP} \tag{3.13}$$

$$recall = \frac{TP}{TP + FN} \tag{3.14}$$

$$f_\beta score = (1 + \beta^2) \cdot \frac{precision \times recall}{\beta^2 \cdot precision + recall} \tag{3.15}$$

where TP, TN, FP and FN are the number of *True Positives*, *True Negatives*, *False Positives* and *False Negatives* respectively.

On the other hand, the *Mean Jaccard* similarity (eq. 3.16) and *Mean per-sample Accuracy* (eq. 3.17) metrics can be used to assess the per-sample performance of a tagging algorithm. In particular, the Jaccard similarity (or index) can be used as a figure of how similar the set of tags associated with a specific sample is with respect to the set of tags predicted for the same sample after binarizing the predictions. The per-sample accuracy is instead defined as the percentage of samples for which the target vector is equal to the prediction vector, i.e. all tags correctly predicted. For an evaluation dataset with M samples the equations of the per-sample evaluation metrics are:

$$\text{Mean Jaccard similarity} = J(T^{(i)}, \hat{T^{(i)}}) = \frac{1}{M}\sum_{i=1}^{M} J(T^{(i)}, \hat{T}^{(i)}) = \frac{1}{M}\sum_{i=1}^{M} \frac{T^{(i)} \cap \hat{T}^{(i)}}{T^{(i)} \cup \hat{T}^{(i)}} \tag{3.16}$$

$$\text{Mean per-sample accuracy} = \frac{1}{M}\sum_{i=1}^{M} \mathbb{I}(\mathbf{y}^{(i)} = \hat{\mathbf{y}}^{(i)}) \tag{3.17}$$

where $T^{(i)}$ is the set of tags associated with sample i, $\hat{T}^{(i)}$ is the set of tags predicted for the same sample after binarizing the predictions, $\mathbf{y} \in \{0, 1\}^T$ is the binary target vector for a PE file (where $\mathbf{y}_n$ indicates whether the n-th tag applies to the file), $\hat{\mathbf{y}}$ is the binarized prediction vector from a given tagging model and $\mathbb{I}$ is the indicator function which is 1 if the condition in the argument is true, 0 otherwise.

3.6.3 Learning from Context: Exploiting and Interpreting File Path Information for Better Malware Detection

Recent static portable executable (PE) malware detection techniques, including [2], typically employ ML-Sec classifiers designed to work with a single numerical feature vector, derived from each file, as input, having as output one or more target labels/tasks. However, as noted by Kyadige et al. in [45], there is still much unused orthogonal information that could be exploited regarding the sample files, such as the file paths. The authors in [45] thus proposed utilizing the static source of contextual information represented by PE file paths as auxiliary data to the classifier in order to augment static ML detectors. File paths, which are already commonly used by malware analysts to correct and investigate detection errors, are available statically with very little overhead, and can seamlessly be integrated into a multi-view static ML detector.

File paths are not inherently malicious or benign; however, given that many malware strains use specifically crafted file paths to perpetrate their malicious intents (a file path may in fact be chosen to increase the odds of the malware being executed, to avoid disk scans, or to hide from

the user's view), they provide much instrumental information that can be used to enhance the overall detection.

In [45], the authors thus proposed the use of a *multi-view* neural network which combines, as input, information about the PE file content, via feature vectors, with information about how likely it is to see such file in a specific location, through file paths, and outputs a detection score.

To compare their results, they actually focused their experiments on three models:

- A baseline file-content-only PE model, which takes only the PE features as input and outputs a malware confidence score.
- Another baseline file-path-only FP model, which takes only the file's paths as input and outputs a malware confidence score.
- Their proposed multi-view PE file-content + contextual file-path (PE + FP) model, which takes both the PE file content features and file paths as inputs, and also outputs a malware confidence score.

The experiments were conducted on a dataset of files and file paths collected from actual scans on customer endpoints from a large anti-malware vendor.

Feature Engineering In order to be able to use file paths in a feed-forward neural network along with PE file content feature vectors, the file paths, which are strings of variable length, needed to be converted into numerical vectors of fixed size. To do this, the authors of [45] created a lookup table keyed on each character with a numeric value (between 0 and the character set size) representing each character. Moreover, the conversion required the file paths to be trimmed to a fixed size, therefore the authors considered just the first 100 characters of each file path.

As features for the content of the PE files, they used floating point 1024-dimensional feature vectors consisting of four distinct feature types, similar to [47]:

1. A 256-dimensional (16x16) 2D histogram of windowed entropy values per byte with a window size of 1024.
2. A 256-dimensional (16x16) 2D logarithmically scaled string length/hash histogram.
3. A 256-dimensional bin of hashes of metadata from the PE header, including PE metadata, including imports, exports, etc.
4. A 256-dimensional (16x16) byte standard deviation/entropy histogram.

In total, they represented each sample as two feature vectors: A PE-content feature vector of 1024 dimensions and a contextual file-path feature vector of 100 dimensions [45].

Inner Workings The model proposed in [45] (fig. 3.5) has two inputs, the 1024 element PE-content feature vector, $\mathbf{x}_{PE}$, and the 100 element file-path integer vector, $\mathbf{x}_{FP}$. The two distinct inputs are fed to two different base sub-networks, each composed by a series of layers with input-specific parameters: θ_{PE} for the PE content part and θ_{FP} for the file-path part. The two sets of parameters (θ_{PE} and θ_{FP}) are jointly optimized during training. The outputs of these base sub-networks are then concatenated and passed through a series of final hidden layers - a joint output sub-network with parameters θ_O terminating with a final dense layer followed by a sigmoid activation function. The final sigmoid activation is used to have as output a detection score between 0 (benign) and 1 (malicious). However, the threshold for determining if a samples is malicious or benign can be set anywhere along the $[0.0, 1.0]$ range according to false positive rate (FPR) and detection rate (TPR) trade offs for the application at hand - a reasonable threshold is typically at or below 10^{-3} FPR.

The PE base sub-network (with parameters θ_{PE}) passes its input $\mathbf{x}_{PE}$ through a series of 5 blocks with sizes 1024, 768, 512, 512, and 512, each consisting of four layers: a Fully Connected

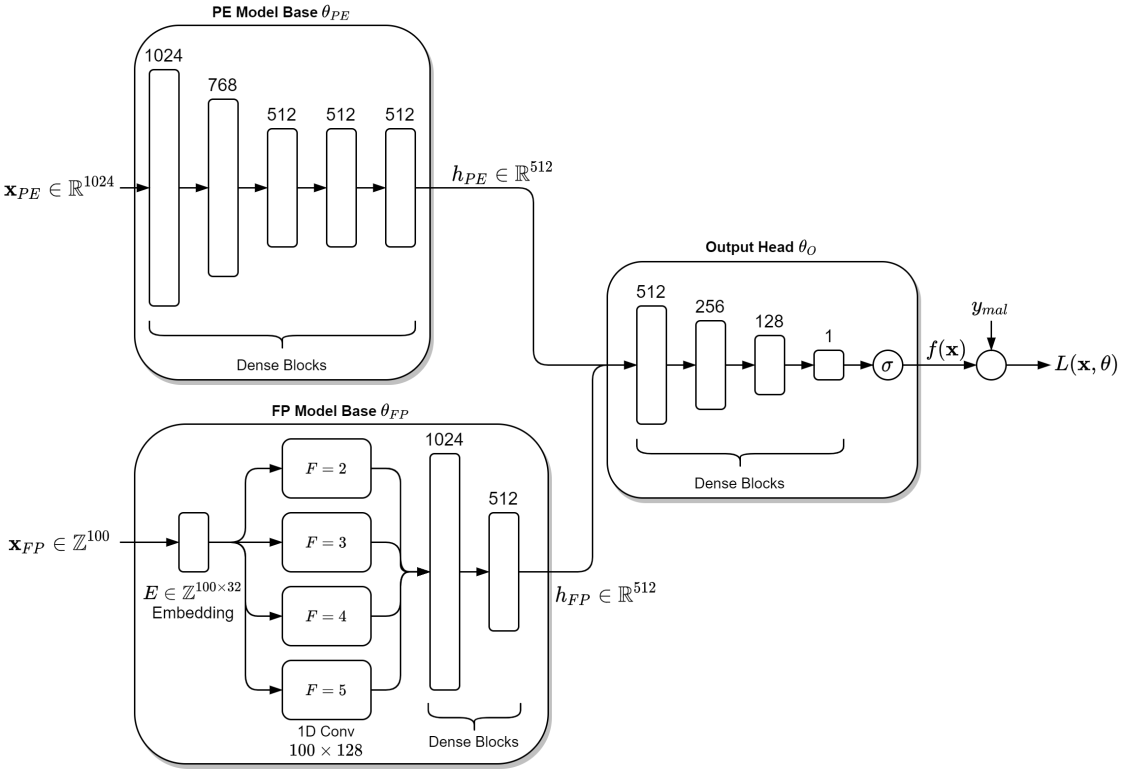


Figure 3.5: PE + FP model architecture

layer, a Normalization layer, a Dropout layer with dropout probability of 0.05, and a Rectified Linear Unit (ReLU) activation function.

The FP base sub-network (with parameters θ_{FP}), on the other hand, passes $\mathbf{x}_{FP}$ first into an Embedding layer that converts the integer input vector into a $(100, 32)$ embedding. This embedding is then fed into 4 parallel convolution blocks with filters of size 2, 3, 4 and 5 respectively, each composed by a 1-D convolution layer with 128 filters, a Layer Normalization layer and a 1-D sum layer to flatten the output to a vector. The flattened outputs of these convolution blocks are then concatenated and serve as input to two dense blocks similar to those found in the PE input arm.

Finally, the outputs from the two base sub-networks are then concatenated and passed into the joint output path with parameters θ_O. This output sub-network is composed by dense blocks (same form as in the PE input arm) of layer sizes 512, 256 and 128. Finally, a single fully-connected layer is employed to project the 128-D output from the previous blocks into a 1-D output, followed by a sigmoid activation function that provides the final output score of the model.

The PE-only model can be easily derived from the PE+FP model removing the FP arm, taking input $\mathbf{x}_{PE}$ and fitting parameters θ_{PE} and θ_O. Similarly, the FP-only model can be constructed by using the PE+FP model but without the PE sub-network, taking input $\mathbf{x}_{FP}$ and fitting parameters θ_{FP} and θ_O. Obviously the first layer of the output sub-network needs to be modified to match the output from the previous layer.

In [45], the authors fit all models using a binary cross entropy loss function. Therefore, denoting the output of the model as $f(\mathbf{x}; \theta)$, where $\mathbf{x}$ is the model input, θ are the model parameters and $y \in \{0,1\}$ are the ground truth labels, the loss is:

$$L(\mathbf{x}, y; \theta) = -y \log\left(f(\mathbf{x}; \theta)\right) + (1 - y) \log\left(1 - f(\mathbf{x}; \theta)\right). \tag{3.18}$$

During training, the equation is optimized for $\hat{\theta}$: the optimal set of parameters that minimize the combined loss over the dataset.

$$\hat{\theta} = \arg\max_{\theta} \sum_{i=1}^{M} L(\mathbf{x}^{(i)}, y_i; \theta), \tag{3.19}$$

where M is the number of samples in the dataset, and y_i and $\mathbf{x}^{(i)}$ are the label and the feature vector of the ith training sample respectively [45].

3.7 Malware Normalization

In order to improve the detection rate of existing anti-malware techniques also against malware produced by advanced packers and toolkits, code *normalization* techniques can be exploited. These techniques consist of a *normalizer* which accepts obfuscated code as input and tries to eliminate obfuscation producing as output the normalized executable. After *normalization*, the usual signature-based techniques can be applied on the normalized sample [22].

Chapter 4

Workflow and Datasets used

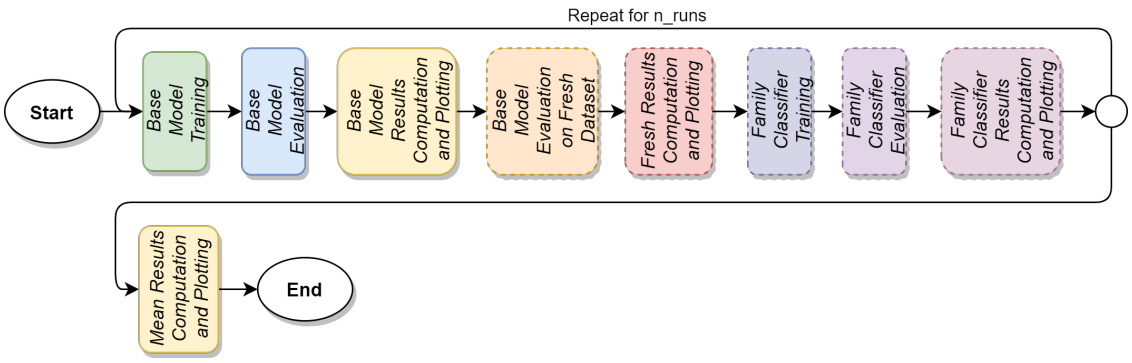


Figure 4.1: Workflow

The workflow presented in image 4.1 summarizes the steps taken throughout the experiments conducted and reported in this document. In particular, for each different model architecture and combination of parameters, the model is first trained and evaluated on the large-scale dataset called ***Sorel 20M dataset*** [6] on the task of SMART tagging and/or malicious/benign label prediction. Next, the model evaluation results are computed and plotted. The model is then evaluated also using a smaller dataset, referred to as '***Fresh dataset***' in this document, specifically crafted for the task of predicting the malware family each specific sample belongs to. The results of this second evaluation are computed and plotted in a separate step. Finally, the fresh dataset is split into training, validation and test subsets and depending on the model type (in particular, in case of the Joint Embedding and the Proposed Model), the learned representation is optionally refined by training (and validating) the architecture resulting by the addition of a family multiclass classifier head on top of the model using the training (and validation) subsets. The resulting classifier is then evaluated on the test subset of the same dataset and the corresponding results are computed and plotted. The whole process presented until now is repeated for *n_runs* times and finally the mean evaluation results are computed, aggregating the results of the single runs.

4.1 Sorel 20M Dataset

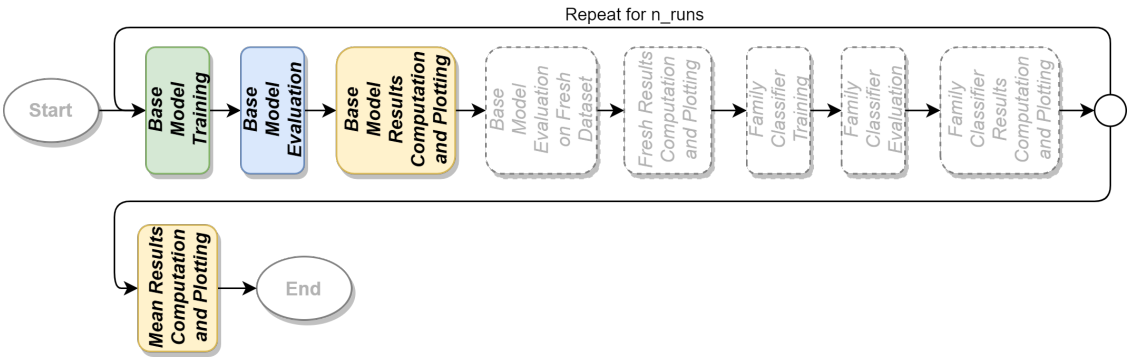


Figure 4.2: Workflow steps which use the Sorel20M dataset

The ***Sorel20M*** large-scale dataset, which was released in 2020 and is used in the workflow steps shown in image 4.2, consists of nearly 20 million files, each represented by the corresponding pre-extracted features, metadata, malicious/benign label derived from multiple sources, vendor detections count information and additional SMART 'tags' to serve as additional targets [6].

SOREL20M dataset was built upon EMBER dataset [48], the first standard and open dataset specifically designed to be used for static malware detection, addressing some of its drawbacks that limited its utility as a malware benchmark set. In particular, EMBER contained 900,000 training samples (300K malicious, 300K unlabelled) and 200,000 test samples (100K malicious, 100K benign) and is therefore of limited size compared to the private/proprietary datasets commercial malware models are usually trained on, which contain from tens to hundreds of millions of samples. Additionally, the small number of validation samples makes it difficult to evaluate the model performance at low false positive rates due to high variance. Finally, the EMBER dataset only provides a single binary malware/benignware label with no additional information.

SOREL20M, on the other hand, provides an order of magnitude more samples for analysis: when using the recommended time splits to establish training, validation and test sets, there are 12,699,013 training samples, 2,495,822 validation samples and 4,195,042 test samples, respectively. Furthermore, Sorel20M [6] provides 9,919,251 samples of malware (7,596,407 training samples, 962,222 validation samples and 1,360,622 test samples), which have been 'disarmed' by setting both the *optional_headers.subsystem* and *file_header.machine* to 0 in order to prevent execution. Additionally, complete PE metadata, obtained via the Python *pefile* module using the *dump_dict* method, is provided for each sample. Finally, the dataset provides for each sample a number of additional targets (SMART tags) for the model that describe behaviour inferred from vendor family labels, together with the vendor detection count and the malicious/benign label.

4.1.1 Sorel 20M Dataset Description

The complete dataset, which is available as a AWS bucket, consists of the following items:

- 9,919,251 original (disarmed) malware samples compressed via the Python *zlib.compress* function
- A SQLite3 and two LMDB databases:
  - The SQLite3 "meta.db" database containing malware labels, tags, detection counts, and first/last seen times
  - The "*ember_features*" LMDB database containing the EMBER features (EMBER features version 2)
  - the "*pe_metadata*" LMDB database containing the PE metadata extracted through the *pefile* module, as described above

- Moreover, some Pre-trained baseline models (a Pytorch feed-forward neural network (FFNN) model and a LightGBM gradient-boosted decision tree model) and their results are also provided, but will not be used in this document.

All samples are identified by their sha256 hash which serves as the primary key for the SQLite3 database, and the key to be used to access the two LMDB databases. LMDB entries are stored as arrays or dictionaries (for Ember feature vectors or PE metadata, respectively) that are then serialized with *msgpack* and compressed with *zlib*.

The data was collected from January 1st, 2017 to April 10th, 2019. In [6], Harang et al. suggest to use the following time-splits of the data (based on the first-seen time in RL telemetry): training data from the beginning of collection until November 29th, 2018; validation data from then until January 12th, 2019; and testing data from January 12th, 2019 to the end of the data.

LMDB database, what is it? Lightning Memory-Mapped Database (LMDB) is a B+trees-based database management library that provides a high-performance embedded transactional database with full ACID semantics in the form of a key-value store (it is not a relational database).

The entire database is exposed in a memory map, and all data fetches return data directly from the mapped memory, so no '*malloc*'s nor '*memcpy*'s occur during data fetches. Therefore, the library is extremely simple as it requires no page caching layer of its own (the OS is responsible for managing the pages), and it is extremely high performant and memory-efficient.

The memory map can be used as a read-only or read-write map. It is read-only by default as this provides total immunity to corruption [49]. LMDB may also be used concurrently in a multi-threaded or multi-processing environment, with read performance scaling linearly by design. In particular it uses shared memory copy-on-write semantics with a single writer; however unlike many similar key-value databases, write transactions do not block readers, nor do readers block writers.

4.1.2 Ember Features

As previously mentioned, Sorel20M dataset consists of a bunch of databases (Sqlite3 and LMDB), indexed by the sha256 hash of the files, which contain the feature vectors representing the samples together with the corresponding labels/metadata. In particular the feature vectors were derived from the samples' code through the use of an open source feature extraction code released by Anderson et al. [48] in 2018 as an attachment to the Endgame Malware BEnchmark for Research (EMBER) dataset. In particular, Sorel20M uses version 2 of this feature extraction code which adds information on data directories to the feature representation with respect to version 1.

The PE file format and the features extracted by the EMBER feature extraction code will be described in the next two sections. Before that, here is a brief description of the EMBER dataset which may be useful in this context.

The EMBER dataset, which was extracted from a large corpus of Windows Portable Executable (PE) malicious and benign files, consists of a collection of JSON lines, where each line contains a single JSON object. Each object includes the following types of data:

- the sha256 hash of the original file as a unique identifier;
- coarse time information (month resolution) about when the file was first seen;
- a malicious/benign label, which may be 0 for benign, 1 for malicious or -1 for unlabelled;
- 8 groups of raw features that include both parsed values as well as format-agnostic histograms.

A notable difference with respect to Sorel20M, however, is that the EMBER dataset is comprised of raw features that are human-readable instead of directly having the feature vectors

required for model building. This was done to allow researchers to decouple raw features from the vectorizing strategy and to improve model interpretability. Anyway, the authors of [48] provided also the code for producing numeric feature vectors from those raw features. The raw feature extraction code and the numeric feature vector generation code were used in succession by Sorel20M authors to directly obtain the feature vector representations from samples.

PE File Format

The Portable Executable (PE) format is a file format for executables, object code, dynamically-linked libraries (DLLs), FON font files and more, used in 32-bit and 64-bit versions of Windows operating systems (it is currently supported on Intel, AMD and variants of ARM instruction set architectures).

The PE format structure consists of a number of standard headers (4.3) followed by one or more sections and it encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code: for example, among other things, it tells the dynamic linker how to map the file into memory. In particular, the PE structure usually consist of the following 3 headers:

- *Common Object File Format (COFF)* file header: it contains important information about the file such as the type of machine for which it is intended, its nature (DLL, EXE, OBJ), the number of sections, the number of symbols, etc.
- *Optional Header*: it can be further divided into:
  - *Standard COFF fields*: they identify the linker version, the size of the code, the size of initialized and uninitialized data, the address of the entry point, etc.
  - *Windows Specific fields*: they provide windows-specific information such as minor and major operating system, subsystem and image versions, stack and heap sizes, section and file alignment, etc.
  - *Data Directories*: they provide pointers to the sections that follow it, which include tables for exports, imports, resources, exceptions, debug information, certificate information, and relocation tables.
- *Section Table*: it outlines the name, offset and size of each section in the PE file.

PE sections, on the other hand, contain code and initialized data that will be mapped at execution time into executable or readable/write-able memory pages, respectively, by the Windows loader, as well as imports, exports and resources defined by the file. Each section contains a header that specifies its size and address. A windows executable typically has the nine predefined sections named *.text*, *.bss*, *.rdata*, *.data*, *.rsrc*, *.edata*, *.idata*, *.pdata*, and *.debug*. Some applications however do not need all of these sections, while others may define still more sections to suit their specific needs.

Among the different sections a PE file may contain here are described some of the most common:

- *Executable code section,* ***.text***: it holds the program code. The *.text* section also contains the code entry point and the *Import Address Table (IAT)* (which is used as a lookup table when the application is calling a function in a different module).
- *Data sections,* ***.bss***, ***.rdata***, ***.data***: The *.bss* section contains uninitialized data for the application, the *.rdata* section contains read-only data. All other application/global variables are stored in the .data section.
- *Resources section,* ***.rsrc***: it contains resource information for a module, such as the ones required for user interfaces: cursors, fonts, bitmaps, icons, menus, etc.
- *Export data section,* ***.edata***: it contains export data for the application or DLL.

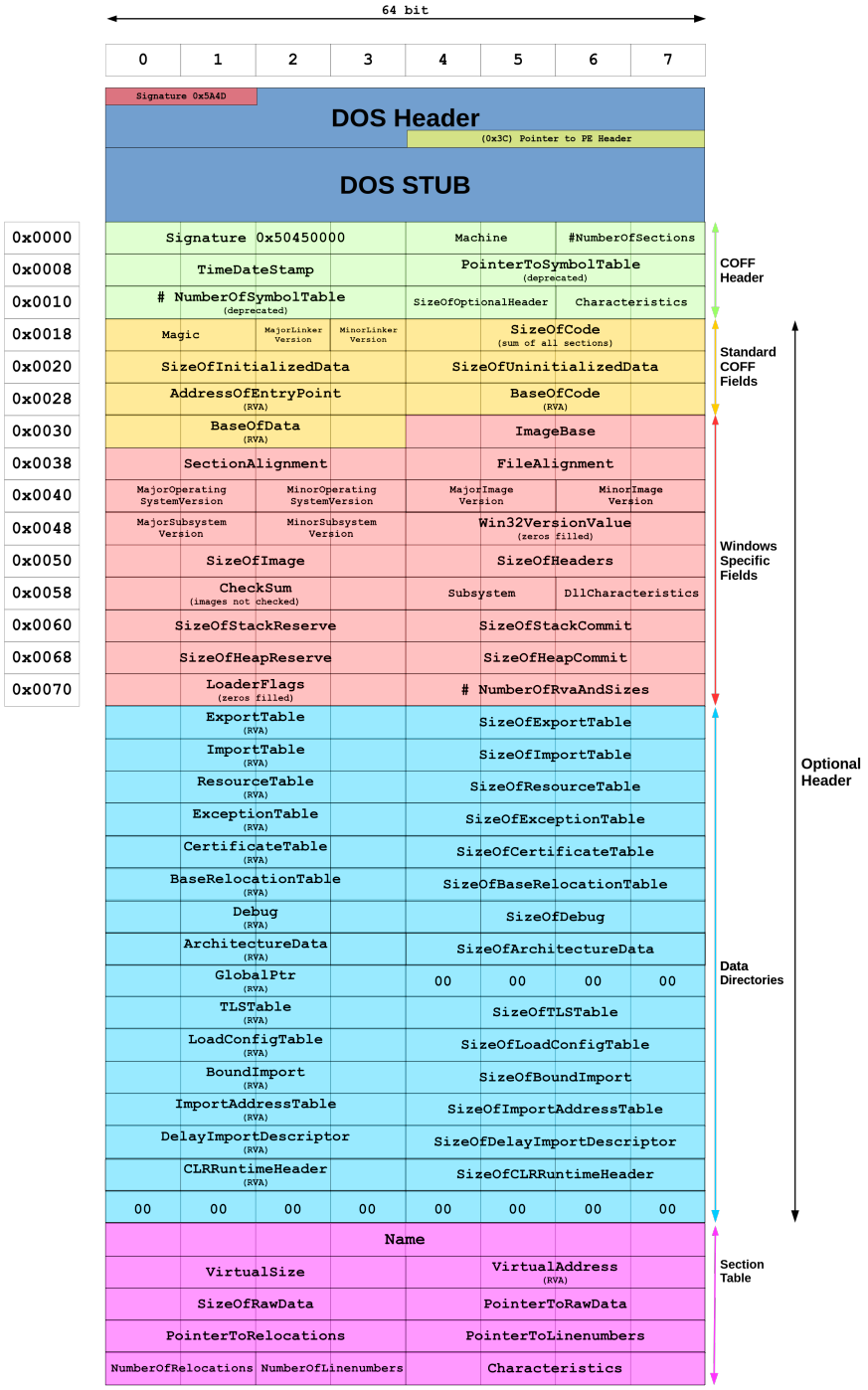


Figure 4.3: PE file structure

- *Import data section,* ***.idata***: it contains import data, including the import directory and import address name table.
- ***.reloc*** *section*: it stores *Relocation tables* which are used by the Windows loader to rebase the PE file if it cannot be loaded at its preferred base address. In fact, PE files normally do

not contain position-independent code but are compiled to a preferred base address, and all addresses emitted by the compiler/linker are fixed ahead of time.

- ***.tls*** *section*: it contains special thread local storage (TLS) structures for storing thread-specific local variables.

A basic PE file normally contains a *.text* code section and one or more data sections (*.data*, *.rdata* or *.bss*).

It is interesting to notice how packers sometimes create new sections, for example, the UPX packer creates a section *UPX1* to store packed data and an empty section *UPX0* that reserves an address range for runtime unpacking [48].

Feature Set Description

The EMBER feature extraction code extracts 8 groups of raw features that include both parsed features and format-agnostic histograms and string counts.

In particular, the EMBER dataset authors [48] made a distinction between human-readable *raw features* and numerical *model features* (or *vectorized features*) derived from the dataset. *Model features* consist of a feature matrix of fixed size used for training models, representing the numerical summary of raw features, wherein strings, imported names, exported names, etc., are captured using the feature hashing trick [50].

Parsed features The first 5 groups of features are extracted after parsing the PE file. Anderson et al. [48] leveraged the *Library to Instrument Executable Formats (LIEF)* [51] as a convenient PE parser. *LIEF* names are used for strings that represent symbolic objects, such as characteristics and properties.

- **General file information** - The set of raw features belonging to the general file information group includes the file size and basic information obtained from the PE header: the virtual size of the file, the number of imported and exported functions, whether the file has a debug section, thread local storage, resources, relocations, and the number of symbols etc.
- **Header information** - From the *COFF* header, the timestamp, the target machine (string) and a list of image characteristics (list of strings) are extracted. From the optional header, instead, the EMBER feature extracting code extracts the target subsystem (string), DLL characteristics (a list of strings), the file magic value as a string (e.g. "PE32"), major and minor image versions, linker versions, system versions and subsystem versions, and the code, headers and commit sizes. Then, in order to create the model features, string descriptors such as DLL characteristics, target machine, subsystem, etc. are summarized using the feature hashing trick, with 10 bins allotted for each indicator vector.
- **Imported functions** - Imported functions and libraries are extracted from the parsing of the import address table. To create numerical features, the feature vector generation code collects the set of unique library names and uses the hashing trick with 256 bins. Similarly, the hashing trick (with 1024 bins) is used to capture individual function names, by representing each as a string in the *library:FunctionName* format (e.g. *kernel32.dll:CreateFileMappingA*).
- **Exported functions** - The raw features include a list of the exported functions. These strings are summarized into model features using the hashing trick with 128 bins.
- **Section information** - Specific section properties are extracted from each section. They include the section name, size, entropy, virtual size, and a list of strings representing the section characteristics. Then the entry point is captured and specified by name. To convert them to model features, the hashing trick is again used on (section name, value) pairs to create vectors containing section size, section entropy, and virtual size (50 bins each). The hashing trick is also used on the entry point characteristics (list of strings).

Format-agnostic features The next 3 groups of features are instead format agnostic, meaning that they do not require the PE file to be parsed for their extraction.

- **Byte histogram** - The byte histogram contains 256 integer values, representing the counts of each byte value within the file. When generating model features, this byte histogram is normalized to a distribution, since the file size is represented as a feature in the general file information.
- **Byte-entropy histogram** - The byte entropy histogram, on the other hand, approximates the joint distribution $p(H, X)$ of entropy H and byte value X. This is done as described by Saxe et al. in [47], by computing the scalar entropy H for a fixed-length window and pairing it with each byte occurrence within the window. This is repeated as the window slides across the input bytes. In particular, EMBER feature extraction code uses a window size of 2048 and a step size of 1024 bytes, with 16×16 bins that quantize entropy and the byte value. These counts are then normalized to sum to 1.
- **String information** - EMBER features also include simple statistics about printable strings (consisting of characters in the range *0x20* to *0x7f*, inclusive) that are at least five printable characters long. In particular, the code extracts information like the number of strings, their average length, a histogram of the printable characters within those strings, and the entropy of characters across all printable strings. In addition, the string feature group also provides the number of strings that begin with specific character sequences such as *C:* (case insensitive) that may indicate a path, *http://* or *http://* (case insensitive) that may indicate a *URL*, *HKEY_* that may indicate a registry key, etc.

Feature vector normalization

When using the Sorel-20m dataset to train and evaluate their example classifier, the authors of [6] first applied a Normalization function on each sample's feature vector. In particular, the function used on each vector component was the one reported in eq. 4.1.

$$f(\mathbf{x}) = \begin{cases} -\log(1-\mathbf{x}) & \text{if } \mathbf{x} < 0 \\ \mathbf{x} & \text{if } \mathbf{x} = 0 \\ \log(1+\mathbf{x}) & \text{if } \mathbf{x} > 0 \end{cases} = \begin{cases} \log(1+\mathbf{x})^{\frac{(1+\frac{\mathbf{x}}{|\mathbf{x}|})}{2}} - \log(1-\mathbf{x})^{\frac{(1-\frac{\mathbf{x}}{|\mathbf{x}|})}{2}} & \text{if } \mathbf{x} \neq 0 \\ x & \text{if } \mathbf{x} = 0 \end{cases} \tag{4.1}$$

This normalization should provide a regularizing effect resulting in better model generalization on unseen samples and was therefore also used for all the experiments conducted and reported in this document.

4.1.3 Improving the Dataset Loading Speed

Dataset Pre-Processing

Harang et al. [6] provided, together with the Sorel20M dataset, the python/pytorch code they used to load it into main memory and pass it as input to their example model. This code, however, is particularly I/O bounded. In fact, in order to load the data related to a single sample i it first gets the corresponding sha256 hash from the Sqlite3 database (which is already loaded in memory) and then uses it as key to access the LMDB database of features. The extracted features are then decompressed, deserialized and normalized. To load the entire dataset, one batch of data of size ***batch_size*** (= 8192 by default) is loaded at a time, repeating this process for ***batch_size*** randomly chosen samples for each batch until the end of the epoch. The code is therefore especially dependant on the hard disk random access speed. The authors of [6] managed to train their model for 10 epochs in under 90 minutes, by training it on an AWS instance with high I/O speeds (exploiting a NVMe SSD), not suffering from any bottleneck. However, when using the less powerful **Colab Pro** instance (which provides a NVIDIA T4 or P100 GPU, 2 CPU cores, 25GB of RAM and 147GB of disk space), not optimized for I/O performance, the time

needed for loading the dataset and subsequentially for training the model increased excessively. This made the use of the original data loading code for the purposes of this project unfeasible.

In order to speed up the dataset loading code the dataset was therefore pre-processed and saved in a easier (and faster) to read format. In particular, the entire dataset was sequentially pre-processed (the features were decompressed, deserialized and normalized) and loaded into a set of 3 parallel memory mapped *numpy* arrays (for the features, labels and sha256 hashes respectively) using the original data loading code, similarly to what was done in the original code at every cycle. The resulting *numpy* arrays were then saved 'as-is' to file. During training (or evaluation) the pre-processed dataset files are read back into memory mapped *numpy* arrays 'as-is' and then used in conjunction with the default Pytorch Dataloader class, with no additional processing needed. This greatly speeded up the overall training. The main drawback of this approach is that the dataset was saved with its features being decompressed, thus taking up a much larger disk portion than the original dataset version.

When using an instance with limited disk size, the amount of space occupied by the dataset is a concern. In particular, the **Colab Pro** instance used for this project had up to $\sim$ 130GB of free disk space and this made it possible to train the models with only approximately half of the samples provided by Sorel20M. In fact for all the conducted experiments only the first 6.000.000 (of 12.699.013) training samples, the first 1.153.846 (of 2.495.822) validation samples and the first 1.846.154 (of 4.195.042) test samples of Sorel20M dataset were considered. The performances of the models presented in this document inevitably suffered from the usage of a smaller dataset during training; this means that the results obtained cannot be directly compared to those of other models which used the entire Sorel-20m dataset. However, the code was designed to work with any dataset size so additional experimentations with more powerful instances can easily be made in the future.

Generator (Dataloader) versions

Pytorch Dataloader In the first implementation of the pre-processed data loading code the original *Pytorch* Dataloader class was used by passing it a specially crafted class called '*Dataset*' (derived from *Pytorch* 'torch.utils.data.Dataset' class) which is responsible for loading the pre-processed version of the dataset and for retrieving the data (features, labels and sha256) for each sample 'i'.

In particular, as it can be seen in algorithms 1 and 2, the *Dataset* class consists of 2 main member functions (although other less important member functions such as the *Len* function are also implemented): *Init* and *GetItem*. In the *Init* constructor function (alg. 1) the dataset, consisting of features (X), labels (y) and sha256 hashes (S), gets loaded into a set of 3 parallel memory mapped arrays. In the *GetItem* member function (alg. 2), instead, the data (features, labels and/or sha256 hash) corresponding to the sample with index '*index*' is retrieved and returned to the function caller (that will be the Pytorch original Dataloader itself). Alternatively, the *GetAsTensors* member function (alg. 3) directly returns the tensors (memory mapped arrays) containing the samples shas, features and labels. This will be later used by the presented alternative generators.

Before model training/evaluation the dataset generator (dataloader) is defined by passing this dataset class to the *Pytorch* Dataloader implementation (*torch.utils.data.Dataloader*) together with additional arguments specifying the batch size, the number of workers and whether to shuffle the data during loading (alg. 4). Then, during training, the *Pytorch* Dataloader class will load the batches of data by iteratively calling the *Dataset*'s *GetItem* function *batch_size* times for each batch concatenating the extracted samples together using the *torch.cat* function.

As previously mentioned, this implementation is faster than the original Sorel20M dataloader code while being also relatively simple; however, given how the pre-processed dataset is saved (consecutive samples are saved on consecutive locations on disk) it is still somewhat inefficient. In fact, the time needed for completing 10 training epochs on a less powerful instance is still unreasonable. One possible optimization would be to read batches in one go as big chunks rather than calling the *GetItem* function on the Dataset once per sample and then concatenating the resulting data. However, random sampling during model training introduces a regularization

Algorithm 1 Dataset class, Init

```
class DATASET
    function INIT(self, ds_root, mode, n_samples, return_shas, ...)
        self.return_shas ← return_shas
        ...
        X_path ← os.path.join(ds_root, 'X_{}_{}.dat'.format(mode, n_samples))
        y_path ← os.path.join(ds_root, 'y_{}_{}.dat'.format(mode, n_samples))
        S_path ← os.path.join(ds_root, 'S_{}_{}.dat'.format(mode, n_samples))

        self.S ← load_as_memmap(S_path, dtype=np.dtype('U64'), mode="r+")
        self.y ← load_as_memmap(y_path, dtype=np.float32, mode="r+")
        self.X ← load_as_memmap(X_path, dtype=np.float32, mode="r+")
    end function
```

Algorithm 2 Dataset class, GetItem

```
    function GETITEM(self, index)
        features ← self.X[index]

        labels ← {}
        labels['malware'] ← self.y[index][0]
        labels['count'] ← self.y[index][1]
        labels['tags'] ← self.y[index][2:]

        if self.return_shas then
            sha ← self.S[index]
            return sha, features, labels
        else
            return features, labels
        end if
    end function
```

Algorithm 3 Dataset class, GetAsTensors

```
    function GETASTENSORS(self)
        if self.return_shas then
            return self.S, self.X, self.y
        else
            return self.X, self.y
        end if
    end function
end class
```

effect which generally improves model generalization on unseen samples, therefore it would be better to find an optimization combining both data loading efficiency and random sampling.

Algorithm 4 Pytorch Dataloader definition

```
ds ← Dataset(ds_root, mode, n_samples, return_shas, ...)
generator ← Pytorch_DataLoader(ds, batch_size, shuffle, n_workers)
```

Generator alt1/alt2 The first attempts at optimizing the generator were inspired by the '***index select***' and '***shuffle in-place***' versions of the ***FastTensorDataLoader*** class suggested in [52]. In particular, the alternative generator 1 ('***alt1***'), again loads the pre-processed dataset as a set of memory mapped arrays and assigns them to a set of tensors (X, y and S) from which batches of data are randomly sampled using the *Pytorch* '***index_select***' function in *multithreading*. In the alternative generator 2 ('***alt2***'), on the other hand, the tensors created by loading the

dataset as done in ***alt1*** are randomly shuffled in place at each iteration. Then the batches of data are extracted (in order) from them in *multithreading*.

Algorithm 5 Alt1/Alt2 FastTensorDataLoader class, Init function

```
1:  function INIT(self, tensors, batch_size, shuffle, n_workers, ...)
2:      self.tensors ← tensors
3:      self.batch_size ← batch_size
4:      self.shuffle ← shuffle
5:      self.n_workers ← n_workers
6:      self.dataset_len ← tensors[0].shape[0]
7:
8:      if num_workers > 1 then
9:          self.async_results ← [ ]
10:         self.pool ← ThreadPool()
11:     end if
12:
13:     self.n_batches ← ceil(self.dataset_len / self.batch_size)
14: end function
```

Algorithm 6 Alt1 FastTensorDataLoader class, Iter function

```
1:  function ITER(self)
2:      if self.shuffle = true then
3:          self.indices ← randperm(self.dataset_len)
4:      else
5:          self.indices ← None
6:      end if
7:
8:      self.i ← 0
9:      return self
10: end function
```

Algorithm 7 Alt2 FastTensorDataLoader class, Iter function

```
1:  function ITER(self)
2:      if self.shuffle = true then
3:          r ← randperm(self.dataset_len)
4:          for i, t ∈ enumerate(self.tensors) do
5:              self.tensors[i] ← t[r]
6:          end for
7:      end if
8:
9:      self.i ← 0
10:     return self
11: end function
```

In practice, both versions of the generator share (for the most part) the *Init* and *Next* functions. The main differences are, instead, in the *Iter* and *GetBatch* functions.

The *Init* function (alg. 5) is used for setting up some ***FastTensorDataLoader*** class variables and optionally initializing the ***ThreadPool*** used for *multithreading*.

The *Iter* function purpose is to set up the parameters for the current cycle at the beginning of each epoch. The two generator alternatives have, however, different *Init* functions. In particular, generator ***alt1*** *Iter* function (alg. 6) resets the count of extracted samples and, if shuffling is enabled, randomly generates the current epoch sample indices (which define the order the samples are extracted from the dataset tensors). On the other hand, generator ***alt2*** *Iter* function (alg. 7) resets the count of extracted samples, and, if shuffling is enabled, it randomly permutes *in-place* the dataset tensors themselves.

Algorithm 8 Alt1/Alt2 FastTensorDataLoader class, Next function

```
function NEXT(self)
    if self.i ≥ self.dataset_len then
        if (self.n_workers = 1 or len(self.async_results) = 0) then
            raise StopIteration
        end if
    end if

    if self.num_workers = 1 then
        batch ← GetBatch(self.tensors, self.batch_size, self.i, (self.indices))
        self.i ← self.i + self.batch_size
        return batch
    else
        while self.i < self.dataset_len and len(self.async_results) < self.n_workers do
            arguments ← (self.tensors, self.batch_size, self.i, (self.indices))
            async_task ← self.pool.apply_async(GetBatch, arguments)
            self.async_results.append(async_task)
            self.i ← self.i + self.batch_size
        end while

        current_result ← self.async_results.pop(0)
        return current_result.get()
    end if
end function
```

Algorithm 9 Alt1 GetBatch function

```
function GETBATCH(tensors, batch_size, i, indices, ...)
    batch ← [ ]
    if indices is provided then
        indices ← indices[i:(i + batch_size)]

        for all t ∈ tensors do
            batch_data ← index_select(t, indices)
            batch.append(batch_data)
        end for
    else
        for all t ∈ tensors do
            batch_data ← t[i:(i + batch_size)]
            batch.append(batch_data)
        end for
    end if

    return batch
end function
```

In order to get a batch of data, the *Next* function (alg. 8), which is similar for both generator versions, is used. In particular, this function monitors the number of samples already extracted from the dataset tensors and raises a *StopIteration* exception when it has cycled through all the data. Moreover, this function is also responsible for extracting and returning batches of data from the dataset tensors. This can be done in sequence or in *multithreading*. Specifically, when the number of workers selected is 1 the function sequentially gets one single batch and returns it after having updated the count of already extracted samples for the current epoch. On the other hand, if more than 1 workers are used the function prepares a number of asynchronous batch extraction tasks to be run in parallel by the threads in the thread pool, then it waits for the first result and returns it. The next call to the function will prepare other tasks and wait for the next first result. The exact number of asynchronous tasks prepared at each call is chosen dynamically to keep a

Algorithm 10 Alt2 GetBatch function

```
1: function GetBatch(tensors, batch_size, i, ...)
2:     batch ← [ ]
3:     for all t ∈ tensors do
4:         batch_data ← t[i:(i + batch_size)]
5:         batch.append(batch_data)
6:     end for
7:
8:     return batch
9: end function
```

fix-sized array of '*num_workers* ' async task results always full.

Finally, in algorithm 9 it is shown the *GetBatch* function used by generator ***alt1***: if the sample *indices* are provided they are used with the *index_select* function to select the data samples for the current batch of data, otherwise a set of consecutive samples is drawn from the dataset. The *GetChunks_batch* function used by generator ***alt2*** (alg. 10) is simpler: it simply selects a set of consecutive samples from the dataset tensors since those were already shuffled if required.

Unfortunately, the presented two generator alternatives did not improve the data loading process enough. More specifically, when considering $6M$ samples, generator ***alt1*** was only slightly faster than the original Pytorch Dataloader, while ***alt2*** was even significantly slower. Generator ***alt1***, in fact, was still fairly similar to the Pytorch Dataloader implementation having used the *index_select* function instead of the slower *torch.cat*. On the other hand, generator ***alt2*** worked by shuffling the entire dataset tensors prior to batch extraction. If this solution surely has the potential of being more efficient for small to moderately sized datasets, it can become a bottleneck in case of huge datasets.

Generator alt3 The generator alternative 3 ('***alt3***') managed to considerably speed up the dataset loading process making a trade off between loading speed and samples dispersion. In fact, the time needed for 1 training epoch (considering 6M training samples) passed from being of approximately 6 hours to $\sim$ 15 minutes.

This optimization uses a new FastTensorDataLoader class which exploits a pool of threads to asynchronously load the dataset into memory in chunks of consecutive data. In particular each thread loads into memory '***n_chunks***' randomly chosen chunks. Each thread then proceeds to concatenate together its '***n_chunks***' chunks, which contain '***chunk_size***' malware samples each, generating each a '***chunk_aggregate***', which is then randomly shuffled and returned by the thread. The ***chunk_aggregates*** are asynchronously inserted in a queue (the order depends on the threads instantiation order) of fixed length equal to the number of workers used. The main dataloader thread manages the ***chunk_aggregate*** queue, instantiating the parallel threads such that the queue is always full. Moreover, the main dataloader thread sequentially extracts, when possible, one ***chunk_aggregate*** at a time from the queue and then proceeds to return one batch of data at a time from it, when required.

More specifically, in the *Init* function (alg. 11), which is similar to the *Init* of the previous alternatives, some ***FastTensorDataLoader*** variables are set/computed, and a ***ThreadPool*** is initialized if necessary.

The *Iter* function (alg. 12), instead, is used to reset the current *chunk aggregate* (and its size) and the number of already processed chunks for the current cycle at the beginning of each *epoch*. Moreover, it also randomly or sequentially (linearly) - depending on whether shuffling is required - initializes the *chunk indices*, which define the order in which the data chunks are retrieved.

In algorithm 13 it is presented the ***FastTensorDataLoader alt3*** *Next* member function, which is used for managing the asynchronous extraction of chunks of consecutive samples from the dataset and for returning a single batch of data at a time. In particular, if the selected number of workers is 1, then, if all the data of the current *chunk aggregate* has been cycled through (or it is the first function call) the function simply extracts a single new *chunk aggregate* (which is

Algorithm 11 Alt3 FastTensorDataLoader class, Init

```
1:  class FastTensorDataloader
2:      function Init(self, tensors, batch_size, chunk_size, chunks, shuffle, n_workers, ...)
3:          self.tensors ← tensors
4:          self.batch_size ← batch_size
5:          self.chunk_size ← chunk_size
6:          self.chunks ← chunks
7:          self.shuffle ← shuffle
8:          self.n_workers ← n_workers
9:          self.dataset_len ← tensors[0].shape[0]
10:
11:         if num_workers > 1 then
12:             self.async_results ← [ ]
13:             self.pool ← ThreadPool()
14:         end if
15:
16:         self.n_batches ← ceil.(self.dataset_len / self.batch_size)
17:         self.n_chunks ← ceil.(self.dataset_len / self.chunk_size)
18:         self.last_chunk_size ← self.dataset_len % self.chunk_size
19:     end function
```

Algorithm 12 Alt3 FastTensorDataLoader class, Iter

```
20:     function Iter(self)
21:         if self.shuffle = true then
22:             self.chunk_indices ← randperm(self.dataset_len)
23:         else
24:             self.chunk_indices ← arange(self.n_chunks)
25:         end if
26:
27:         self.chunk_agg ← None
28:         self.chunk_agg_size ← 0
29:         self.chunk_i ← 0
30:         return self
31:     end function
```

a concatenation of multiple chunks selected randomly or linearly depending on the value of the *self.shuffle* variable) through the **GetChunks** function. The number of already processed chunks for the current *epoch* and the number of extracted samples from that specific *chunk aggregate* are then updated/reset accordingly. This is however done only if the number of chunks already processed is less than the total amount of chunks, otherwise the *StopIteration* exception is raised. If, on the other hand, the number of workers is greater than 1, the function prepares a number of asynchronous *chunk aggregate* extraction tasks such that the queue (of size equal to the number of workers selected) in which they are inserted is always full, until the number of extracted chunks is less than the total amount. It then updates the number of extracted chunks appropriately. Then, when the current *chunk aggregate* data has been explored completely (or if it is the first function call), if the async task queue is empty, meaning there are no more *chunk aggregates* for the current *epoch*, it raises the *StopIteration* exception; otherwise the function gets the first async task from the queue and waits for its result (which is the extracted *chunk aggregate*). Independently from the number of workers used, the function then proceeds to efficiently get a single batch of data from the current already-in-memory *chunk aggregate* and returns it.

The **GetBatch** function used in generator ***alt3*** is the same one already exploited in version ***alt2*** (alg. 10); however, generator ***alt3*** sees the addition of the **GetChunks** function (alg. 14) which is used to extract a number of chunks of consecutive data from the dataset and combine them into an *aggregate chunk*. In particular, the function first computes the size of the resulting *chunk aggregate* depending on the number of chunk indices specified and the chunk size (considering as

Algorithm 13 Alt3 FastTensorDataLoader class, Next

```
    function NEXT(self)
        if self.n_workers = 1 then
            if self.chunk_agg is None or self.i ≥ self.chunk_agg_size then
                if self.chunk_i ≥ self.n_chunks then
                    raise StopIteration
                end if

                start_i ← self.chunk_i
                end_i ← start_i + self.chunks
                arguments ← (self.tensors, self.chunk_indices[start_i:end_i],
                             self.chunk_size, self.last_chunk_size,
                             self.n_chunks, self.shuffle)
                self.chunk_agg, self.chunk_agg_size ← GetChunks(arguments)
                self.chunk_i ← end_i
                self.i ← 0
            end if
        else
            while self.chunk_i < self.n_chunks and len(self.async_results) < self.n_workers do
                start_i ← self.chunk_i
                end_i ← start_i + self.chunks
                arguments ← (self.tensors, self.chunk_indices[start_i:end_i],
                             self.chunk_size, self.last_chunk_size,
                             self.n_chunks, self.shuffle)
                async_task ← self.pool.apply_async(GetChunks, arguments)
                self.async_results.append(async_task)
                self.chunk_i ← end_i
            end while

            if self.chunk_agg is None or self.i >= self.chunk_agg_size then
                if len(self.async_results) = 0 then
                    raise StopIteration
                end if

                current_result ← self.async_results.pop(0)
                self.chunk_agg, self.chunk_agg_size ← current_result.get()
                self.i ← 0
            end if
        end if

        batch ← GetBatch(self.chunk_agg, self.batch_size, self.i)
        self.i ← self.i + batch[0].shape[0]
        return batch
    end function
end class
```

a special case the last chunk of the dataset which may have a smaller size). Then, the *chunk aggregate* itself is initialized as a series of properly sized empty tensors (one for samples features, one for labels and one for sha256 hashes) which are later filled sequentially by the data chunks got from the dataset tensors. Finally, the *chunk aggregate* data gets shuffled in place, if needed, and returned.

As it can be seen from the algorithms provided, this version of the generator depends on the value of a number of parameters among which the most important ones are ***chunk_size*** and ***n_chunks***. Indeed, selecting different values for ***chunk_size*** and ***n_chunks*** has an impact on the speed of the generator and on the samples dispersion: the samples are not anymore randomly chosen from the whole dataset, but from a random sub-part of it (the ***chunk_aggregate***). This

Algorithm 14 Alt3 GetChunks function

```
1: function GetChunks(tensors, chunk_indices, chunk_size, last_chunk_size, n_chunks, shuffle)
2:     if n_chunks −1 in chunk_indices then
3:         chunk_agg_size ← (len(chunk_indices) −1) × chunk_size + last_chunk_size
4:     else
5:         chunk_agg_size ← len(chunk_indices) × chunk_size
6:     end if
7:
8:     chunk_agg ← [ ]
9:     for all t ∈ tensors do
10:        chunk_agg.append(emptyTensor)
11:    end for
12:
13:    c_start ← 0
14:    for all idx ∈ [0, ..., len(chunk_indices)] do
15:        t_start ← chunk_indices[idx] × chunk_size
16:        if chunk_indices[idx] ≠ n_chunks −1 then
17:            c_end ← c_start + chunk_size
18:            t_end ← t_start + chunk_size
19:        else
20:            c_end ← c_start + last_chunk_size
21:            t_end ← t_start + last_chunk_size
22:        end if
23:
24:        for i, t ∈ enumerate(tensors) do
25:            chunk_agg[i][c_start:c_end] ← t[t_start:t_end]
26:        end for
27:        c_start ← c_end
28:    end for
29:
30:    if shuffle then
31:        r ← randperm(chunk_agg_size)
32:        for i, t ∈ enumerate(chunk_agg) do
33:            chunk_agg[i] ← t[r]
34:        end for
35:    end if
36:
37:    return chunk_agg, chunk_agg_size
38: end function
```

effectively decreases the amount of dispersion (and randomness) of batches, possibly affecting the final model generalization. It is therefore better to consider a higher value for ***n_chunks*** possibly decreasing ***chunk_size***. Furhtermore, the values for ***chunk_size*** and ***n_chunks***) should be chosen in conjunction since their product results in the number of samples loaded into main memory (RAM) at once for a single worker thread. Increasing too much this number (together with the number of workers used) can potentially saturate the main memory (RAM) of the instance used.

Generator parameters optimization To understand the behaviour of the FastTensorDataLoader ***alt3*** on the target instance and choose the best combination of values for the parameters ***chunk_size*** and ***n_chunks*** when using 8 threads (which is double the amount of cores available in the instance used for this project), the code speed was cross evaluated using powers of 2 for both values. In particular the data loading code was evaluated using values for ***chunk_size*** and ***n_chunks*** got from two intervals and the corresponding average *speed* and average *elapsed time* heatmap plots were generated.

In practice ***chunk_size*** and ***n_chunks*** values were chosen to be the powers of 2 ranging from 2^4 to 2^{14} (included) and from 2^3 to 2^{13}, respectively. Moreover, in order to constrain the evaluation to meaningful values only, two additional parameters have to be set: ***min_mul*** and ***max_mul***. In fact, as previously mentioned, the product of ***chunk_size*** $\times$ ***n_chunks*** gives the total number of samples in one ***chunk_aggregate*** residing in main memory; this quantity must be constrained to a certain range to avoid using too much RAM while being able to retrieve at least one *batch* of data from the *chunk aggregate*. The parameters ***min_mul*** and ***max_mul*** are used exactly to indicate the minimum and maximum number of batches retrievable from the resulting ***chunk_aggregates*** and therefore indirectly pose a constrain on the product '***chunk_size*** $\times$ ***n_chunks***'.

The data loading code (generator ***alt3***) performance was evaluated setting ***min_mul*** to a value of 1, while ***max_mul*** was set to 32. Those values were chosen in order to always be able to retrieve at least one *batch* from the ***chunk_aggregate*** without saturating the RAM available on the instance used for this project (in fact, having for ***max_mul*** a value greater than 32 would mean having 32$\times$ ***batch_size*** samples loaded into memory at the same time, per worker, which was too much for the available instance).

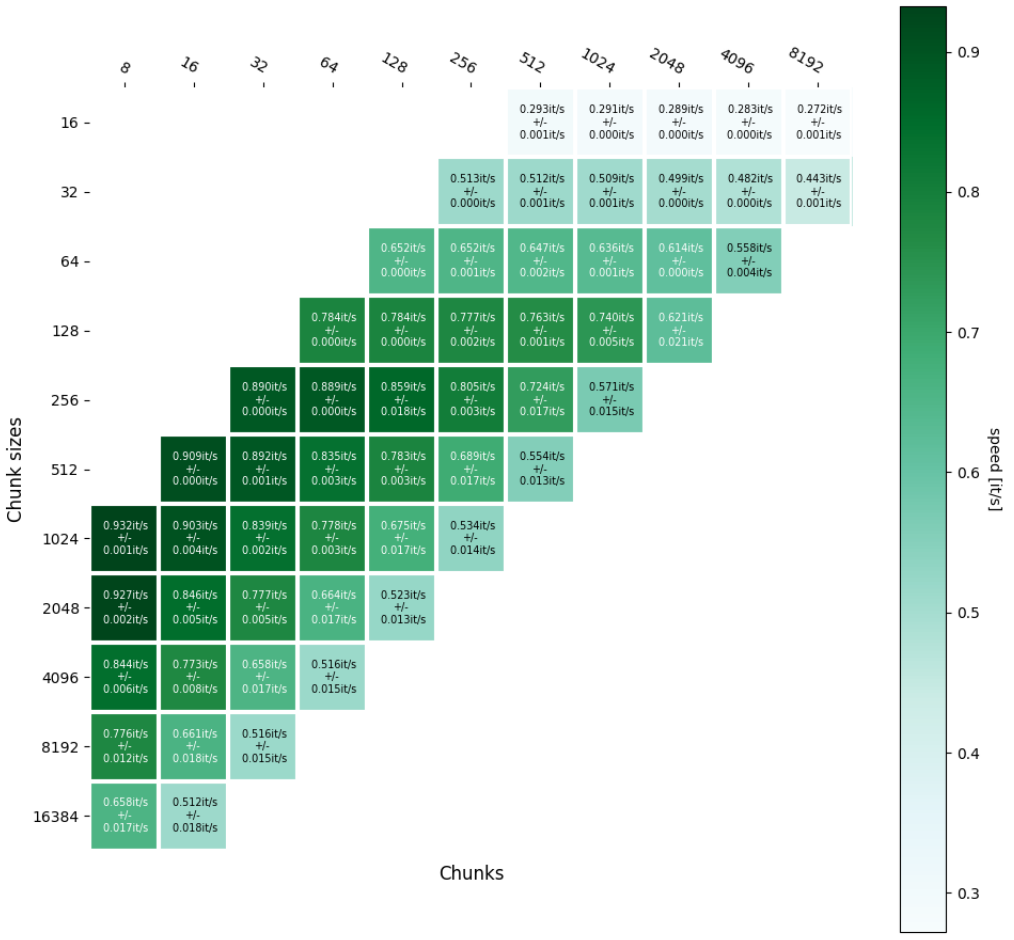


Figure 4.4: Generator Alt.3 Speeds Heatmap, Higher is Better

Given the resulting heatmaps (4.4, 4.5), for all the experiments conducted for this project ***chunk_size*** and **n_chunks** were permanently set to **256**, which seemed to be a good compromise between data loading speed and samples dispersion.

In any case, with respect to the previous solutions (original *Pytorch* dataloader and the generators ***alt1*** and ***alt2***) the speed up was huge. This new version took, in fact, approximately 15 minutes, on average, to complete 1 training epoch while the previous solution took, at best, $\sim$ 6 hours. Therefore this alternative generator was the one selected to be used for loading the dataset in all the conducted experiments. However, it has to be reiterated that this generator

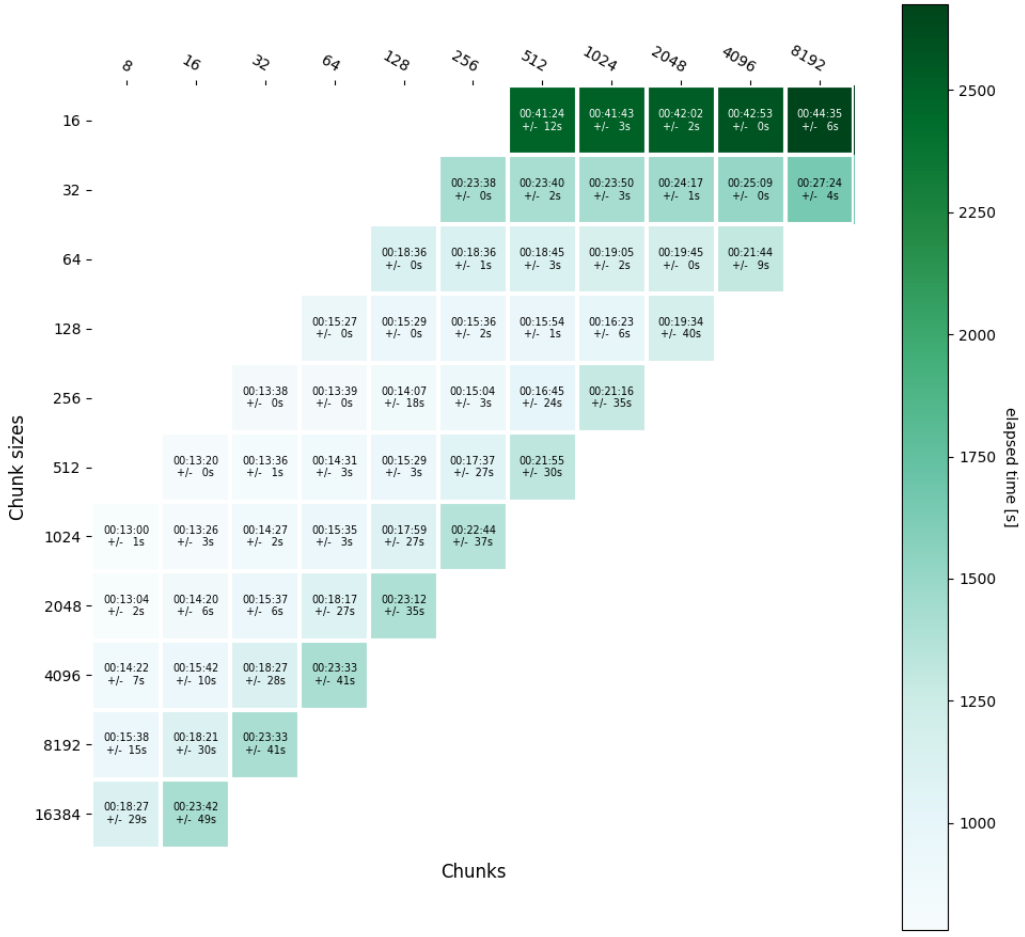


Figure 4.5: Generator Alt.3 Elapsed Times Heatmap, Lower is Better

version works by approximating the random sampling step during training which in turn may hinder the model generalization on unseen samples. The results presented in this document may therefore be slightly lower than what could be achieved on a more powerful instance using the original *Pytorch* dataloader.

4.2 Fresh Dataset

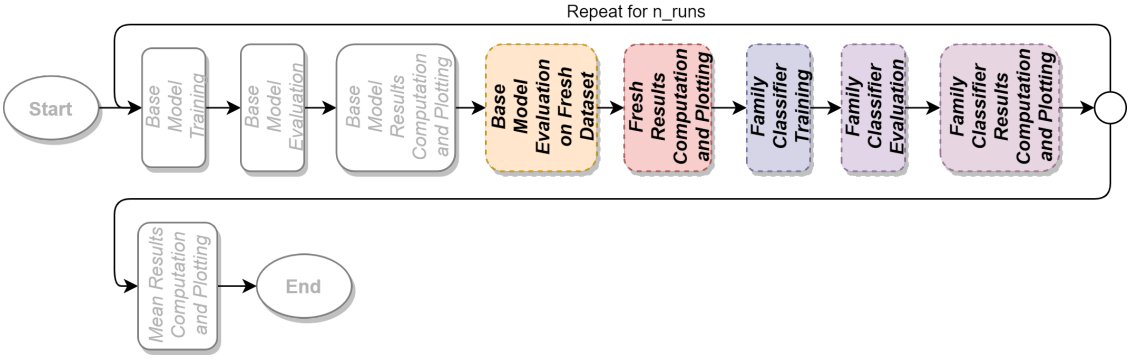


Figure 4.6: Workflow steps which use the Fresh dataset

In order to evaluate the learned representation of PE files (embedding) of the proposed model on the sample family prediction and ranking tasks (fig. 4.6), a further dataset, also referred to as ***fresh_dataset*** throughout this document (and in the code), was created. It consisted of a number of sample files' feature vectors along with the corresponding family labels and sha256 hashes.

More specifically, a number '$\boldsymbol{m}$' of PE malware families were selected from the list of the most prominent ones present in Italy at the time of writing (as reported by CERT-AGID summary [53]) of which Malware Bazaar provided at least '$\boldsymbol{x}$' sample files. Then, '$\boldsymbol{x}$' sample files per family were downloaded by Malware Bazaar [54] along with their metadata information (label, sha256 hash etc.) and the corresponding numerical feature vectors were extracted using the EMBER (version 2.0) feature extraction and numerical feature generation codes [48] (and Sorel-20m normalization). Malware Bazaar is a malware sample database maintained by malware analysts which provides examples of malware executables and high-quality manually crafted classifications/descriptions for different malware families and it is therefore a good source for creating the new (fresh) dataset.

A function called ***build_fresh_dataset*** was specifically implemented with the purpose of creating the ***fresh_dataset***, given the following arguments:

- a list of '$\boldsymbol{f}$' malware families (with $\boldsymbol{f} \geq \boldsymbol{m}$) in order of importance;
- the number '$\boldsymbol{x}$' of samples per family to download;
- the number '$\boldsymbol{m}$' of families to consider.

More specifically, the function downloads '$\boldsymbol{x}$' samples per malware family from Malware Bazaar (if possible, otherwise it skips the malware family and considers the next in order of importance between the ones selected) ordering them by the time they were first seen (from most to least recent), it extracts the features from each sample and creates the new dataset containing '$\boldsymbol{n} = \boldsymbol{x} \times \boldsymbol{m}$' samples which are then stored on disk as (*numpy*) memory mapped arrays (as it was done with the pre-processed Sorel20M dataset). Therefore, between the $\boldsymbol{f} \geq \boldsymbol{m}$ malware families provided to the function, only the first $\boldsymbol{m}$ for which $\boldsymbol{x}$ PE samples can be retrieved from Malware Bazaar are considered.

In particular, the ***fresh dataset*** created for this project consists of $\boldsymbol{x} = \mathbf{1.000}$ samples per family, for a total of **10.000** PE samples, considering the following $\boldsymbol{m} = \mathbf{10}$ families: *FormBook*, *AgentTesla*, *Gozi*, *Quakbot*, *Loki*, *Heodo*, *RemcosRAT*, *TrickBot*, *GuLoader* and *AveMariaRAT*.

4.2.1 Fresh Dataset Generator (Dataloader) implementation

Algorithm 15 Fresh Dataset class, Init

```
1:  class DATASET
2:      function INIT(self, S, X, y, sig_to_label_dict, return_shas)
3:          self.S ← S
4:          self.X ← X
5:          self.y ← y
6:          self.N ← len(S)
7:          self.sig_to_label_dict ← sig_to_label_dict
8:          self.n_families ← len(sig_to_label_dict.keys())
9:          self.return_shas ← return_shas
10:         self.sig_to_label_inv_dict ← {v: k ∀ k, v ∈ self.sig_to_label_dict.items()}
11:     end function
```

The dataset class used for opening the fresh dataset - shown in alg. 15, 16, 17 and 18 - is similar to the one used for the Sorel-20m dataset. This version, however, is also instantiatable with an already opened dataset/subset providing the corresponding tensors. The 15 function, in fact, instantiates the dataset given a set of already loaded dataset tensors (containing the shas,

Algorithm 16 Fresh Dataset class, FromFile

```
    function FromFile(cls, ds_root, return_shas)
        ndim ← 2381
        X_path ← os.path.join(ds_root, 'X_fresh.dat')
        y_path ← os.path.join(ds_root, 'y_fresh.dat')
        S_path ← os.path.join(ds_root, 'S_fresh.dat')
        state_to_label_path ← os.path.join(ds_root, 'sig_to_label.json')

        S ← load_as_memmp(S_path, dtype=np.dtype('U64'), mode='r+')
        y ← load_as_memmap(y_path, dtype=np.float32, mode='r+')
        X ← load_as_memmap(X_path, dtype=np.float32, mode='r+')
        return cls(S, X, y, sig_to_label_dict=sig_to_label_dict, return_shas=return_shas)
    end function
```

Algorithm 17 Fresh Dataset class, GetItem

```
    function GetItem(self, index)
        features ← self.X[index]

        label ← self.y[index]

        if self.return_shas then
            shas ← self.S[index]
            return sha, features, label
        else
            return features, label
        end if
    end function
```

Algorithm 18 FreshDataset class, GetAsTensors

```
    function GetAsTensors(self)
        if self.return_shas then
            return self.S, self.X, self.y
        else
            return self.X, self.y
        end if
    end function
end class
```

features and labels), while the 16 function opens the dataset, loads it as as set of memory mapped arrays and finally calls the class Init function.

Algorithm 19 shows how the fresh dataset generator (Dataloader) gets instantiated. This dataloader instantiation is similar to the one previously presented when loading the Sorel-20m dataset through the Pytorch dataloader with the addition of the possibility to split the dataset into training, validation and test sub-sets. In particular, if the dataset split proportions are provided (in the form of a list containing 3 integer values), the dataset is split into training, validation and test subsets by first opening the entire fresh dataset and then splitting it by using the purposefully created **TrainValidTestSplit** function, presented in alg. 20. On the other hand, if the split proportions are not provided, the entire fresh dataset is opened from file without being split.

The **TrainValidTestSplit** (alg. 20) function's purpose is that of splitting the tensors passed as input into 3 subsets, following the given proportions. This is done by first computing the number of samples, per family, to select for each of the sub-splits by multiplying the respective proportion by the Fresh Dataset total number of samples per family. The function then computes the sub-splits indices assigning the first '*n_samples*['test']' samples (per family) of the original dataset (which are the most recent ones) to the test sub-split, the following '*n_samples*['valid']'

Algorithm 19 Fresh Dataloader definition

```
if splits is not None and len(splits) = 3 then
    ds ← Dataset(ds_root, return_shas)
    splits_sum ← sum(splits)
    for all i ∈ range(len(splits)) do
        splits[i] ← splits[i] / splits_sum
    end for

    S, X, y ← ds.GetAsTensors()
    S_train, S_valid, S_test, X_train, X_valid, X_test, y_train, y_valid, y_test ←
        TrainValidTestSplit(S, X, y, proportions=splits,
                                     n_samples_tot=len(ds),
                                     n_families=ds.n_families)
    train_generator ← Pytorch_DataLoader(Dataset(S_train, X_train, y_train, ...), ...)
    valid_generator ← Pytorch_DataLoader(Dataset(S_valid, X_valid, y_valid, ...), ...)
    test_generator ← Pytorch_DataLoader(Dataset(S_test, X_test, y_test, ...), ...)
    generator ← (train_generator, valid_generator, test_generator)
else
    ds ← Dataset.FromFile(ds_root, return_shas)
    generator ← Pytorch_DataLader(ds, ...)
end if
```

Algorithm 20 Train Valid Test Split function

```
function TrainValidTestSplit(*tensors, proportions, n_samples_tot, n_families)
    n_samples_per_family ← n_samples_tot // n_families
    n_samples ← {'test': math.floor(proportions[2] × n_samples_per_family),
                 'valid': math.floor(proportions[1] × n_samples_per_family),
                 'train': math.ceil(proportions[0] × n_samples_per_family)}
    indices ← {}
    for all i ∈ range(n_families) do
        if i = 0 then
            start ← 0
            for all k, v ∈ n_samples.items() do
                end ← start + v
                indices[k] ← np.arange(start, end)
                start ← end
            end for
        else
            start ← i × n_samples_per_family
            for all k, v ∈ n_samples.items() do
                end ← start + v
                indices[k] ← np.concatenate(indices[k], np.arange(start, end))
                start ← end
            end for
        end if
    end for

    rv ← []
    for all t ∈ tensors do
        rv.append(t[indices['train']])
        rv.append(t[indices['valid']])
        rv.append(t[indices['test']])
    end for
    return rv
end function
```

samples (per family) to the validations sub-split and the remaining ones (which are the least recent ones) to the training sub-split. Finally, the input tensors are split by indexing them using the previously computed indices.

4.2.2 Base Model Evaluation with Fresh Dataset

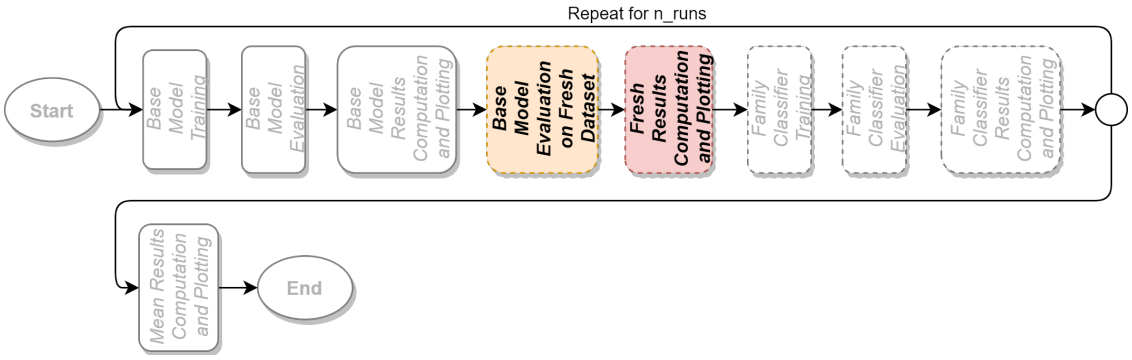


Figure 4.7: Base Model Evaluation with Fresh dataset Workflow steps

The actual model evaluation on the ***fresh dataset*** is computed by another function called ***EvaluateFresh***. This function evaluates the model learned representation in both the f-way malware family classification and malware family ranking tasks.

f-way Malware Family Classification task evaluation The model learned representation first gets evaluated on the task of *f-way* family classification via nearest neighbour search, using the fresh dataset. In particular a function called *EvaluateFreshScores* randomly samples k files per family as anchor samples, and q files per family as query samples. Each of the $(f \times q)$ query samples is then predicted to belong to the same class as its closest anchor sample in feature space (Joint Embedding space in case of the Joint Embedding and Proposed Models). The number q of query samples - per family - considered during the actual evaluation was set to 23 and the number of anchor samples k was varied from 1 to 10 (included). The sampling process (for both anchors and query samples), classification and evaluation was repeated 15 times per value of k to obtain uncertainty estimates for the results. More specifically, the *accuracies* and the values of a number of other binary cross-entropy *macro* (per-class average) and *micro* (global average) statistics such as *precision*, *recall* and *f1-score* were collected for each number of anchors k and the resulting trends were plotted (with uncertainty estimates). Moreover, the function saves to file also the confusion matrixes corresponding to the best and worst classification accuracy results for the number k of anchors which produced the overall best accuracy. Finally, also the per-family, micro (global average) and macro (per-family average) ROC-AUC scores are computed for each number k of anchors and the resulting trends are plotted (with uncertainty).

Malware Family Ranking task evaluation On the other hand, to evaluate the model learned representation on the family ranking task, the function first randomly selects '$\boldsymbol{q}$' query samples among the ones in the $fresh\_dataset$. Then, it computes the similarity (using the model's *GetSimilarity* function) between each query sample with the other '$n-1$' samples present in the $fresh\_dataset$ ordering the latter by similarity. At this point, for each query sample there is a ranking of all the other $n-1$ samples sorted from the most to least similar. Using those rankings the function finally calculates the **MRR** (*Mean Reciprocal Rank*) and **MAP** (*Mean Average Precision*) scores for the model learned representation.

The Mean Reciprocal Rank (**MRR**, 4.3) is the average of the Reciprocal Ranks (**RR**, 4.2) of a series of queries. In particular, the Reciprocal Rank (**RR**) of a query response is the multiplicative inverse of the rank (position in the ranking) of the single sample belonging to the family of interest

which classified with the highest rank (it is the closest).

$$RR(i) = \frac{1}{rank_i} \tag{4.2}$$

$$MRR = \frac{1}{q}\sum_{i=1}^{q} RR(i) = \frac{1}{q}\sum_{i=1}^{q} \frac{1}{rank_i} \tag{4.3}$$

where $\boldsymbol{rank_i}$ refers to the rank position of the first relevant sample (meaning the first sample with the same family as the current query sample) for the $\boldsymbol{i}$-th query and $\boldsymbol{q}$ is the number of queries.

The **MAP** score (4.6), on the other hand, takes into account all the samples in the family of interest, with proper weights, instead of considering just the best classified one. Therefore, a model will have a higher **MRR** score if it classifies a single sample of the family of interest higher in the ranking; by contrast, it will have a higher **MAP** score if it classifies all the samples belonging to the family of interest higher in the ranking.

$$P(k) = \frac{|\text{Relevant Samples Retrieved @}k|}{k} \tag{4.4}$$

$$AvgP(k) = \sum_{k=1}^{n} \frac{P(k) \times rel(k)}{\text{Number of Relevant Samples}} \tag{4.5}$$

$$MAP = \frac{1}{q}\sum_{i=1}^{q} AvgP(i) \tag{4.6}$$

where $\boldsymbol{P(k)}$ denotes the *precision* at position k in the ranking, $\boldsymbol{AvgP(k)}$ indicates the *average precision* at position k in the ranking, $\boldsymbol{n}$ is the number of samples in the ***fresh_dataset***, $\boldsymbol{rel(k)}$ is an indicator function which equals 1 if the sample item at rank $\boldsymbol{k}$ is a relevant sample for the current query, zero otherwise, and finally $\boldsymbol{q}$ is the number of queries. Moreover, a sample is considered relevant if its family is the same as the current query sample one.

The ***EvaluateFresh*** function provides, together with the **MRR** and **MAP** scores, the complete rankings for the q query samples in a single *json* file. Moreover, for convenience, 4 particular rankings are also saved as *csv* files. These 4 rankings are those that produced:

- the maximum **RR** (Reciprocal Rank);
- the minimum **RR** (Reciprocal Rank);
- the maximum **AP** (Average Precision);
- the minimum **AP** (Average Precision).

For all the experiments presented in this document the number of query samples q was set to 100.

4.2.3 Family Classifier Training and Evaluation

The Fresh Dataset - which contains 10.000 samples of malicious PE files belonging to 10 different malware families - provides, as previously mentioned, the information regarding the malware family each of its samples belongs to. When split into training, validation and test subsets, it can also be used to directly train, validate and evaluate a Malware Family Classifier model.

When trying to improve the relatively poor results of the **Proposed Model** in the malware Family prediction task, the 10.000 samples of the Fresh Dataset were divided following a 7-1-2 ratio: the training subset was created selecting the first 700 samples, per family, and therefore consisted of 7.000 samples. The validation and test subsets, instead, respectively consisted of 1.000 and 2.000 samples (100 and 200 samples per family, respectively). The resulting dataset splits were then used to train (and evaluate) both a malware family classifier built on top of the **Proposed**

Model (using *transfer learning*) and a separate comparable malware family classifier trained from scratch. In particular, the training and validation subsets were used for training the classifier and validating its accuracy, while the test subset served for evaluating the classifier accuracy (and other classification scores) on unseen samples. The resulting multi-class classification statistics, which included *accuracy*, *Jaccard Similarity score*, *recall*, *precision*, *f1-score* were then computed and plotted together with the resulting model *AUC-ROC* scores and confusion matrixes.

In particular, the *Jaccard Similarity score*, *recall*, *precision*, *f1-score* and *AUC-ROC* scores were computed by averaging the respective class scores in 3 different ways:

- **micro** average: calculates the metrics globally by counting the total true positives, false negatives and false positives.
- **macro** average: calculates the metrics for each class (family) label, and finds their unweighted mean (thus not taking class imbalance into account).
- **weighted** average: calculates the metrics for each class (family) label, and finds their average, weighted by support (the number of true instances for each label) effectively altering the 'macro' average to account for label imbalance.

Chapter 5

Previous Methods

The state of the art malware detection/description methods presented above (section 3.6), namely *Auxiliary Loss Optimization for Hypothesis Augmentation* (**ALOHA** model, [2]), *Automatic Malware Description via Attribute Tagging and Similarity Embedding* (**Joint Embedding** model, [3]) and *Learning from Context: Exploiting and Interpreting File Path Information for Better Malware Detection* (**File content + contextual file path (PE + FP)** model, [45]), represent a good starting point for further research in this topic. Moreover, the models used in those methods can easily be modified to work with Sorel20M dataset, with the exception of the **PE + FP** model [45] given that the mentioned dataset provides no file path information (and it is unfeasible to extend it with the needed additional information).

Both the **ALOHA** [2] and **Joint Embedding**, [3] models were therefore implemented in *Python/Pytroch* following the descriptions in the respective papers and having as a starting point the example code of Sorel20M dataset.

5.1 Implementation

Both the implementations developed for this project only loosely followed the original models' structure as described in the respective papers. Here are presented the actual implementation details.

5.1.1 ALOHA model

The ALOHA model implementation developed for this project (fig. 5.1) consists of a shared base topology and a number of output heads. The shared base topology is composed of 5 blocks, each consisting of Dropout, a dense layer, a batch normalization layer, and an Exponential Linear Unit (ELU) activation function, with 1024, 768, 512, 512, and 512 hidden units respectively. The exact number of layers, along with the layers' dimensions, the dropout probability, the normalization function (layer normalization or batch normalization) and the activation function (ELU, $ReLU$, $PReLU$ or $LeakyReLU$) used, however, are dynamically set in the code at network definition, thus providing a good amount of customizability. The shared model base, given an input sample feature vector $\mathbf{x}$ (of size 2381) got from Sorel20M dataset, outputs an intermediate representation $h = f(\mathbf{x})$ of size 512 which is then used as input to the different parallel output heads.

The output heads used in this implementation are similar to those described in the original ALOHA paper ([2]) with the exclusion of the *Per-Vendor Malicious/Benign* prediction head which could not be reproduced given the lack of the needed information (namely the per-vendor malicious/benign labels) in the Sorel20M dataset. In particular, for each output head (*Malicious/Benign label*, *Vendor Count* and *Malicious tags* prediction heads) an additional block - consisting of one or more dense layers and activation functions - was appended to the shared base topology. Moreover, the *Vendor Count* and *Malicious tags* prediction heads were made optional, meaning that they can be turned on or off dynamically in the code at network definition.

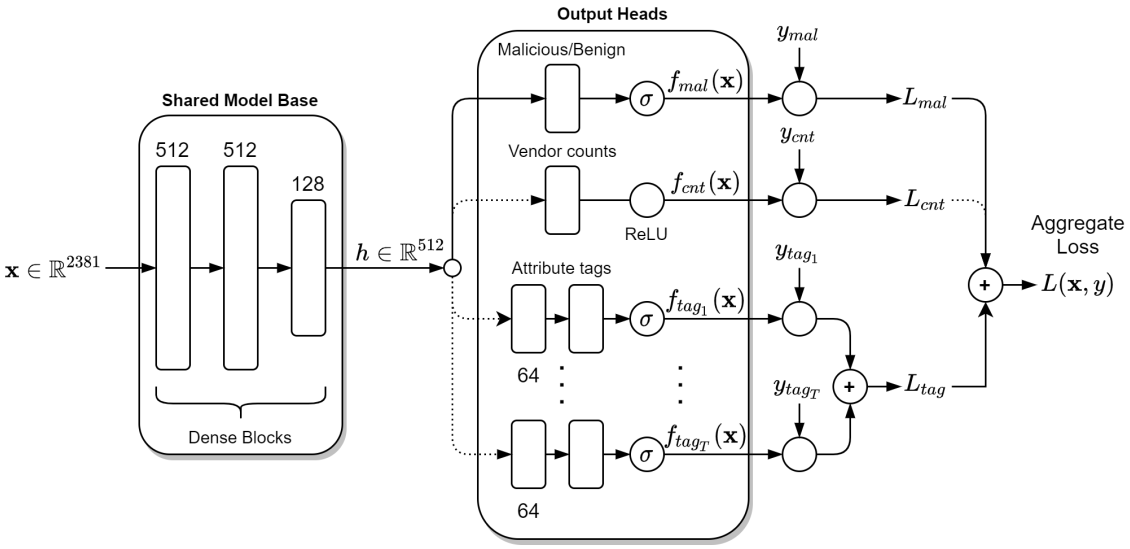


Figure 5.1: ALOHA model implementation architecture

More specifically, for the *Malicious/Benign* label prediction head, which overall applies function $f_{mal}(x)$ to each sample feature vector $\mathbf{x}$, a single dense layer followed by the *sigmoid* activation function was used. The binary cross entropy loss L_{mal} between the resulting $f_{mal}(x)$ and the ground truth y_{mal} is then computed exactly as described in paper [2] (and as presented in paragraph 3.6.1, eq. 3.1).

The *Vendor Count* output head, on the other hand, is composed of a single dense layer followed by the *ReLU* activation function, and outputs $f_{cnt}(x)$. Again, the negative log likelihood loss L_{cnt} with respect to the ground truth value y_{cnt} is computed as described by Rudd et al. [2] (and as presented in paragraph 3.6.1, eq. 3.3).

Finally, the *Malicious (SMART) tags* output head consists of three additional dense layers of size 64, 64 and *n_tags* (11) respectively, interleaved by the *ELU* activation function and followed by a final *sigmoid* non linearity. The resulting structure therefore has $T = 11$ parallel paths, one for each tag, that simultaneously compute the functions $f_{tag_i}(\mathbf{x}), \forall i \in 1, .., 11$. The aggregate tag loss L_{tag} is then computed, as described in paper [2] (and as presented in paragraph 3.6.1, eq. 3.5), as the sum of the individual tag binary cross entropy losses $L_{tag_i}, \forall i \in 1, .., 11$ with respect to each individual ground truth tag y_{tag_i}.

The final aggregate loss $L(\mathbf{x}, y)$ is computed as the weighted sum of all the output heads losses, as previously described in paragraph 3.6.1 (eq. 3.6).

Net Definition

Algorithms 21, 22 and 23 show some pseudo-code describing how the *ALOHA* model was actually implemented using *Python* language and *Pytorch* library. In particular alg. 21 describes how the the model is constructed/initialized: first the model shared base topology is defined as a sequence of layers composed each of a *Pytorch* linear layer, a normalization layer (which can be chosen to be either *Layer Normalization* or *Batch Normalization*), an activation function dynamically chosen at network instantiation among the ones provided by *Pytorch* library, and Dropout. Then, the *malware* (*Malicious/Benign*) head is defined as a single *Pytorch* linear layer followed by a *Sigmoid* activation. The *Vendor Count* head, on the other hand is simply composed of a single linear layer followed by a Rectified Linear Unit (*ReLU*) non linearity. Finally, the *SMART Tag* prediction head consists of a sequence of 3 linear layers interleaved by *ELU* activations and trailed by a final *Sigmoid* activation function.

Alg. 22, instead, describes how the input data (PE file numerical feature vectors of size = 2381) is forwarded through the network to produce the final predictions. In particular, the input feature

Algorithm 21 ALOHA Net class, Init

```
class NET
    function INIT(self, use_malware, use_counts, use_tags, n_tags, feature_dimension, layer_sizes, dropout_p, activation_function, normalization_function)

        self.use_malware ← use_malware
        self.use_counts ← use_counts
        self.use_tags ← use_tags
        self.n_tags ← n_tags
        layers ← [ ]

        if layer_sizes is None then
            layer_sizes ← [512, 512, 128]
        end if

        for all i, ls ∈ enumerate(layer_sizes) do
            if i = 0 then
                layers.append(nn.Linear(feature_dimension, ls))
            else
                layers.append(nn.Linear(layer_sizes[i −1], ls))
            end if

            layers.append(normalization_function(ls))
            layers.append(activation_function())
            layers.append(nn.Dropout(dropout_p))
        end for

        self.model_base ← nn.Sequential(*tuple(layers))

        self.malware_head ← nn.Sequential(nn.Linear(layer_sizes[-1], 1), nn.Sigmoid())
        self.count_head ← nn.Sequential(nn.Linear(layer_sizes[-1], 1), nn.ReLU())
        self.tag_head ← nn.Sequential(nn.Linear(layer_sizes[-1], 64), nn.ELU(),
                                      nn.Linear(64, 64), nn.ELU(),
                                      nn.Linear(64, n_tags), nn.Sigmoid())
    end function
```

Algorithm 22 ALOHA Net class, Forward

```
    function FORWARD(self, data)
        rv ← {}
        base_out ← self.model_base(data)

        if self.use_malware then
            rv['malware'] ← self.malware_head(base_out)
        end if

        if self.use_counts then
            rv['count'] ← self.count_head(base_out)
        end if

        if self.use_tags then
            rv['tags'] ← self.tag_head(base_out)
        end if

        return rv
    end function
```

Algorithm 23 ALOHA Net class, Compute Loss

```
    function ComputeLoss(predictions, labels, loss_wts)
        if loss_wts is None then
            loss_wts ← {'malware': 1.0, 'count': 0.1, 'tags': 1.0}
        end if

        loss_dict ← {'total': 0.}
        if 'malware' ∈ labels then
            malware_loss ← Binary_cross_entropy(predictions['malware']), labels['malware'])

            if 'malware' ∈ loss_wts then
                weight ← loss_wts['malware']
            else
                weight ← 1.0
            end if

            loss_dict['malware'] ← malware_loss
            loss_dict['total'] ← loss_dict['total'] + malware_loss × weight
        end if

        if 'count' ∈ labels then
            count_loss ← PoissonNLLLoss(predictions['count']), labels['count'])

            if 'count' ∈ loss_wts then
                weight ← loss_wts['count']
            else
                weight ← 1.0
            end if

            loss_dict['count'] ← count_loss
            loss_dict['total'] ← loss_dict['total'] + count_loss × weight
        end if

        if 'tags' ∈ labels then
            tags_loss ← Binary_cross_entropy(predictions['tags'], labels['tags'])

            if 'tags' ∈ loss_wts then
                weight ← loss_wts['tags']
            else
                weight ← 1.0
            end if

            loss_dict['tags'] ← tags_loss
            loss_dict['total'] ← loss_dict['total'] + tags_loss × weight
        end if

        return loss_dict
    end function
end class
```

vector first passes through the shared base topology to produce an intermediate representation *base_out*, which is then used as input for all the output heads to produce their predictions.

Finally, in alg. 23 it is shown how the loss between the model predictions and the actual labels gets computed. In particular, the final total loss is the weighted sum of the losses of the different output heads which are enabled for the current run. Both the *Malicious/Benign* and the *SMART tags* prediction heads use the binary cross entropy loss function to compute the loss between their

predictions and the ground truth labels. On the other hand, the *Vendor Count* prediction head uses the PoissonNLLLoss (Negative log likelihood loss with Poisson distribution) function.

5.1.2 Joint Embedding

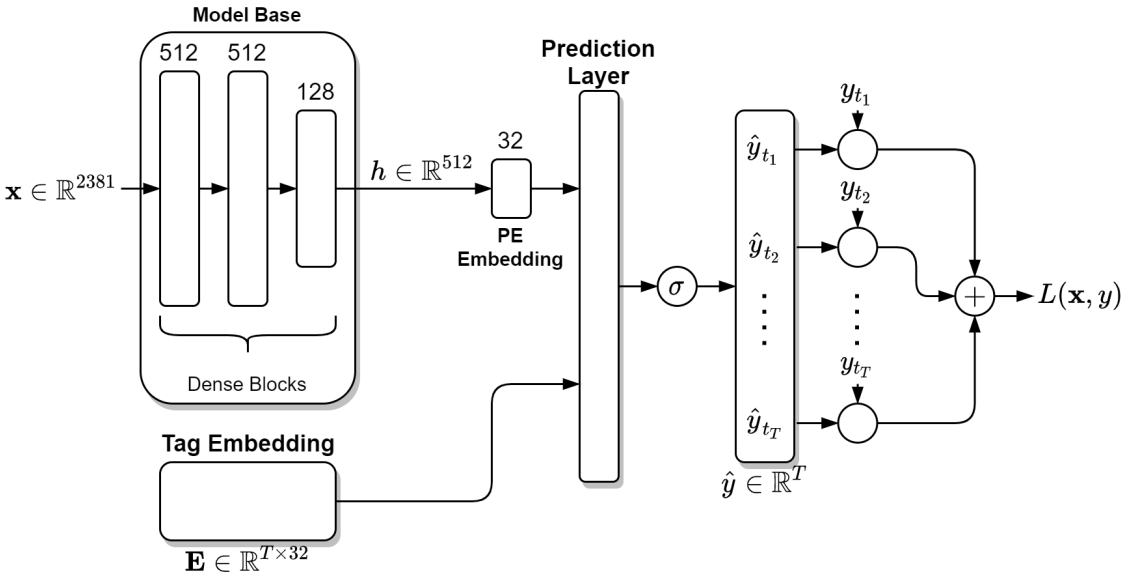


Figure 5.2: Joint Embedding model implementation architecture

The implementation of the Joint Embedding model developed for this project (fig. 5.2) is composed of 3 main parts: the *Model Base/PE Embedding* topology, the *Tag Embedding* and the *Prediction Layer*.

The model *base topology* consists of a series of 5 dense blocks, each composed by Dropout, a linear layer, a Batch Normalization layer and ELU (Exponential Linear Unit) activation function, of output sizes 1024, 768, 512, 512 and 512 respectively. Similarly to the ALOHA implementation, at network definition it is again possible to dynamically set the exact number of layers, together with the linear layers' sizes, the dropout probability, the actual normalization function to use between *Batch Normalization* and *Layer Normalization* and the activation function (between *ELU*, *ReLU*, *PReLU* and *LeakyReLU*) for the base topology. Anyway, the model base topology outputs an intermediate representation h of size 512 given an input sample feature vector $\mathbf{x}$ (of size 2381). This intermediate representation is then used as input to a further linear layer with output size equal to the chosen ***Joint Embedding*** size (set to 32 throughout all the experiments). The output of this further layer is the one used as input to the *Prediction Layer* and corresponds to the representation of the input file features x in the Joint Embedding space.

On the other hand, the *Tag Embedding* matrix $\mathbf{E} \in \mathbb{R}^{T \times 32}$, which is the same as the original implementation [3], maps each tag t_n to the corresponding representation in the Joint Embedding space and is therefore used as second input to the *Prediction Layer*.

Finally, the *Prediction Layer* combines the sample (PE) and tags embeddings producing a similarity score matrix that is run through a *sigmoid* activation function (by value) resulting in the probabilities of each tag t being associated to sample $\mathbf{x}$. The function used to compute the similarity score between the two embedding vectors can be chosen dynamically at network definition between: the *dot product* (eq. 5.1), the *cosine similarity* (eq. 5.2) or the inverse of the *Euclidean distance*. In particular, there are 3 versions of inverted *Euclidean distance* that can be selected at network definition: *exp* (eq. 5.3), *inv* (eq. 5.4) or *inv_pow* (eq. 5.5).

$$\text{dot product}(\mathbf{a}, \mathbf{b}) = \mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{D} a_i b_i \tag{5.1}$$

$$\text{cosine similarity}(\mathbf{a},\mathbf{b}) = \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|} = \frac{\sum_{i=1}^{D} a_i b_i}{\sqrt{\sum_{i=1}^{D} a_i^2}\sqrt{\sum_{i=1}^{D} b_i^2}} \tag{5.2}$$

$$\text{euclidean similarity exp}(\mathbf{a},\mathbf{b}) = e^{\left(\frac{-d(\mathbf{a},\mathbf{b})}{\alpha}\right)} \tag{5.3}$$

$$\text{euclidean similarity inv}(\mathbf{a},\mathbf{b}) = \frac{1}{1+\frac{d(\mathbf{a},\mathbf{b})}{\alpha}} \tag{5.4}$$

$$\text{euclidean similarity inv\_pow}(\mathbf{a},\mathbf{b}) = \frac{1}{1+\frac{d(\mathbf{a},\mathbf{b})^2}{\alpha}} \tag{5.5}$$

where $d(\mathbf{a},\mathbf{b})$ is the Euclidean distance between points $\mathbf{a}$ and $\mathbf{b}$ as defined in eq. 5.6 and α is a multiplicative factor that can be arbitrarily set at network definition when using one of the inverted *Euclidean distance* functions. In the final Joint Embedding model implementation α is set to 1.0.

$$d(\mathbf{a},\mathbf{b}) = \sqrt{\sum_{i=1}^{D}(a_i - b_i)^2} \tag{5.6}$$

The predicted tag probabilities outputted by the *Prediction Layer* are then used to compute the respective binary cross entropy losses with respect to the ground truth tags $y_{t_i}, \forall i \in 1,..,11$, exactly as described in paper [3] (and presented in paragraph 3.6.2, eq. 3.11), which are combined together to form the final loss $L(\mathbf{x}, y)$.

Net Definition

Algorithm 24 (***Joint Embedding* model init function**) shows how the ***Joint Embedding*** model was implemented in *Python*/*Pytorch* code. More specifically, it describes how the network is initialized/defined: first a **base topology** is defined similarly to the one used in the ***ALOHA*** model implementation previously described (a series of linear layers interleaved by normalization layers, activation functions and dropout). The ***PE embedding*** sub-network is, on the other hand, defined as a simple linear layer with output size equal to the chosen *embedding_dimension*, followed by a Normalization layer and an activation function. Finally, the ***Tags embedding*** is defined as an embedding matrix of size (*n_tags*, *embedding_dimension*), constraining the maximum norm of each tag embedding to *max_embedding_norm* (set to 1 throughout the experiments).

Alg. 25 (***Joint Embedding* model forward function**), instead, describes how the input data (the PE files' numerical feature vectors) is forwarded through the network to produce the final tag predictions. In particular, the input data is first fed into the model base topology to produce an intermediate representation ***base_out***. This transient output is then used as input for the ***PE Embedding*** topology to produce the samples' representation in the joint embedding space (*pe_embedding*). On the other hand, the tags embeddings (one per tag) are extracted from the *tags_embedding* matrix. The **similarity scores** between the samples' and tags' embeddings are computed via a simple ***dot product*** (implemented through a matrix multiplication) between the embeddings. Finally the tag probabilities are obtained by mapping the similarity scores to the range [0,1] through the *sigmoid* function.

In alg. 26 (***Joint Embedding* model forward function with *cosine similarity***) is presented an alternative to the aforementioned **Model forward function** that uses the ***Cosine similarity*** instead of the ***dot product*** to compute the similarity scores. For the most part this version is equal to the previous one but for how the similarity scores are computed. In fact, in this case they correspond to the ***Cosine similarities*** between the ***pe_embedding*** and the ***tags_embedding***, which are values in the range $[-1, 1]$, mapped to values in the $[0, 1]$ interval. Moreover, in this case, since the similarity scores are already in the $[0, 1]$ range, the tag probabilities coincide with the similarity scores.

Algorithm 24 Joint Embedding Net class, Init

```
class NET
    function INIT(self, use_malware, use_counts, use_tags, n_tags, feature_dimension, embedding_dimension, max_embedding_norm, layer_sizes, dropout_p, activation_function, normalization_function)

        self.n_tags ← n_tags
        self.embedding_dimension ← embedding_dimension
        layers ← [ ]

        if layer_sizes is None then
            layer_sizes ← [512, 512, 128]
        end if

        for all i, ls ∈ enumerate(layer_sizes) do
            if i = 0 then
                layers.append(nn.Linear(feature_dimension, ls))
            else
                layers.append(nn.Linear(layer_sizes[i −1], ls))
            end if

            layers.append(normalization_function(ls))
            layers.append(activation_function())
            layers.append(nn.Dropout(dropout_p))
        end for

        self.model_base ← nn.Sequential(*tuple(layers))
        self.sigmoid ← nn.Sigmoid()

        self.pe_embedding ← nn.Sequential(
            nn.Linear(layer_sizes[-1], self.embedding_dimension),
            normalization_function(self.embedding_dimension),
            activation_function())
        self.tags_embedding ← nn.Embedding(self.n_tags, self.embedding_dimension,
                                           max_norm=max_embedding_norm)
    end function
```

Algorithm 25 Joint Embedding Net class, Forward (with *Dot Product*)

```
    function FORWARD(self, data)
        rv ← {}
        base_out ← self.model_base(data)

        pe_embedding ← self.pe_embedding(base_out)
        tags_embedding ← self.tags_embedding(Dataset.encoded_tags)

        similarity_scores ← torch.matmul(pe_embedding, tags_embedding.T)
        rv['similarity'] ← similarity_scores
        rv['probability'] ← self.sigmoid(similarity_scores)
        return rv
    end function
```

A further alternative to the **Model forward function** which uses one of the *Inverted Pairwise Euclidean Distances* proposed in sec. 5.1.2 instead of the *dot product*, is presented in alg. 27 (***Joint Embedding* model forward function with *Inverted Pairwise Euclidean Distance***). Again, this alternative is pretty much equal to the original forward function (alg. 25). In this case, however, the similarity scores are produced by inverting, through the use of the

Algorithm 26 Joint Embedding Net class, Forward (with *Cosine similarity*)

```
46:  function FORWARD(self, data)
47:      ...
48:      similarity_scores ← torch.div(torch.add(Cosine_similarity(
49:          pe_embedding, tags_embedding, dim=2), 1.0), 2.0)
50:      rv['similarity'] ← similarity_scores
51:      rv['probability'] ← similarity_scores
52:      return rv
53:  end function
```

Algorithm 27 Joint Embedding Net class, Forward (with *Inverted Pairwise Euclidean Distance*)

```
54:  function FORWARD(self, data)
55:      ...
56:      distances ← torch.cdist(pe_embedding, tags_embedding, p=2.0)
57:      similarity_scores ← DistanceToSimilarity(distances, a=sim_function_a,
58:                                               function=sim_function)
59:      rv['similarity'] ← similarity_scores
60:      rv['probability'] ← similarity_scores
61:      return rv
62:  end function
```

Algorithm 28 Joint Embedding Net class, GetEmbedding

```
63:  function GETEMBEDDING(self, data)
64:      rv ← {}
65:      base_out ← self.model_base(data)
66:      pe_embedding ← self.pe_embedding(base_out)
67:      rv['embedding'] ← pe_embedding
68:      return rv
69:  end function
```

Algorithm 29 Joint Embedding Net class, GetSimilarity (with *Dot Product*)

```
70:  function GETSIMILARITY(self, first_embedding, second_embedding)
71:      similarity_scores ← torch.matmul(first_embedding, second_embedding.T)
72:
73:      return {'similarity': similarity_scores, 'probability': self.sigmoid(similarity_scores)}
74:  end function
```

Algorithm 30 Joint Embedding Net class, GetSimilarity (with *Cosine similarity*)

```
75:  function GETSIMILARITY(self, first_embedding, second_embedding)
76:      similarity_scores ← torch.div(torch.add(F.cosine_similarity(
77:          first_embedding, second_embedding), 1.0), 2.0)
78:
79:      return {'similarity': similarity_scores, 'probability': similarity_scores}
80:  end function
```

specially crafted **DistanceToSimilarity** function presented in alg. 33 (which simply applies the selected distance inversion formula among the ones mentioned in sec. 5.1.2), the euclidean distances between the samples' and the tags' embeddings in the joint latent space.

Alg. 28 (***JointEmbedding* model get embedding function**) presents the *GetEmbedding* function which is used to extract the PE embedding representation corresponding to the input file feature vector. This is particularly useful when performing different kinds of model performance evaluations together with the **GetSimilarity function**, which instead returns the similarity between two different input embeddings. Of course, depending on the similarity function selected at model definition the *GetSimilarity* function has to be different. In particular, if the **dot product**

Algorithm 31 Joint Embedding Net class, GetSimilarity (with *Inverted Pairwise Euclidean Distance*)

```
    function GetSimilarity(self, first_embedding, second_embedding)
        distances ← torch.cdist(first_embedding, second_embedding, p=2.0)
        similarity_scores ← DistanceToSimilarity(distances, a=self.embedding_dimension,
                                                 function=sim_function)

        return {'similarity': similarity_scores, 'probability': similarity_scores}
    end function
```

Algorithm 32 Joint Embedding Net class, Compute Loss

```
    function ComputeLoss(predictions, labels, loss_wts)
        loss_dict ← {'total': 0.}

        similarity_loss ← Binary_cross_entropy(
            predictions['probability'], labels['tags'], reduction='none').sum(dim=1).mean(dim=0)

        loss_dict['jointEmbedding'] ← similarity_loss
        loss_dict['total'] ← similarity_loss

        return loss_dict
    end function
end class
```

Algorithm 33 Distance-to-Similarity function

```
function DistanceToSimilarity(distance, a, function)
    if function = 'exp' then
        similarity ← torch.exp(torch.div(distances, −a))

    else if function = 'inv' then
        similarity ← torch.pow(torch.add(torch.div(distances, a), 1.0), -1.0)

    else if function = 'inv_pow' then
        similarity ← torch.pow(
            torch.add(torch.div(torch.pow(distances, 2.0), a), 1.0), -1.0)

    else
        raise Exception
    end if

    return similarity
end function
```

is selected the function, shown in algorithm 29 (***JointEmbedding* model get similarity function with Dot Product**), simply computes the similarity scores as the dot product between the two input embeddings and then defines the probabilities by further applying the sigmoid activation function on those scores. On the other hand, if the **cosine similarity** is selected, the function (30 - ***JointEmbedding* model get similarity function with Cosine Similarity**) assigns to both similarity scores and resulting probabilities the normalized (to the [0,1] range) cosine similarity between the two embeddings. Finally, in case the **Inverted Pairwise Euclidean Distance** is selected, the function (31 - ***JointEmbedding* model get similarity function with Inverted Pairwise Euclidean Distance**) first computes the euclidean distance between the two input embeddings and then assigns to both similarity scores and resulting probabilities the inversion of such distances through the use of the *DistanceToSimilarity* function.

Finally, in algorithm 32 it is described how the loss, which guides the training procedure, is

computed. In particular, it corresponds to the **binary cross entropy loss** between the predicted tag probabilities and the actual tag labels, as depicted by the formulas mentioned in paragraph 3.6.2.

5.2 Experiments

Both the models were trained (and validated) for 10 *epochs* each on the first $6M$ training and 1.153.846 validation samples of the pre-processed Sorel20M dataset (loaded with generator ***alt3***) using different numbers of layers, numbers of nodes per layer (layer sizes), activation functions, normalization functions and dropout values for the model base topology. Moreover, various combinations of optimizers (**Adam** or **SGD**), learning rates (LR), momentum, weight decay and loss weights were used in separate training runs. Moreover, in the case of the **Joint Embedding** model also different similarity score functions such as the cosine similarity and the inverse of the Euclidean distance were tested instead of the dot product. The models which performed best during validation were then evaluated on the first 1.846.154 test samples of Sorel20M dataset looking for their tag and/or malicious/benign label (present only in the ALOHA model implementation) prediction binary cross entropy statistics and mean per-sample scores such as *TPR*, *accuracy*, *recall*, *precision*, *f1* score, *jaccard similarity* and *mean-per-sample-accuracy* at FPR = 1% as well as the resulting tag and/or malicious/benign *ROC* curves and *AUC* scores.

The final **ALOHA** architecture, which had a base topology consisting of 5 layers of sizes 1024, 768, 512, 512 and 512 respectively, with dropout probability $p_d = 5\%$ and **ELU** activation function, in particular, was trained with a learning rate $LR = 10^{-3}$, **Adam** optimizer, no momentum nor weight decay and loss weights set to 1.0 for the malicious/benign head, 0.1 for the count head and 0.1 for the tag head (as described in the original paper [2]).

On the other hand, the base topology of the final **Joint Embedding** architecture consisted of 5 layers of sizes 1024, 768, 512, 512, and 512 respectively, with dropout probability $p_d = 5\%$ and **ELU** activation function. The overall model was trained with a learning rate $LR = 10^{-3}$, **Adam** optimizer, no momentum nor weight decay. Moreover, the final model uses the dot product as similarity score between tags' and samples' embeddings.

The *ALOHA* and *Joint Embedding* model implementations evaluation results are reported in chapter 7.

5.3 Training and Evaluation algorithms

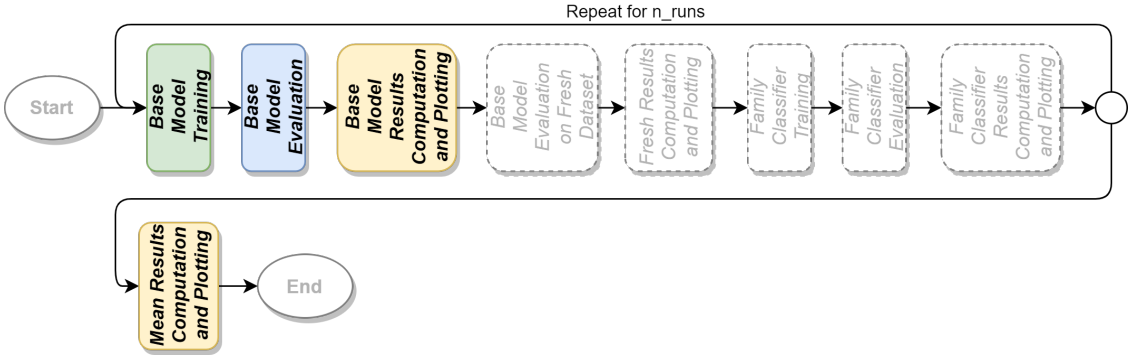


Figure 5.3: Model Training and Evaluation Workflow steps

Both the ***ALOHA*** and the ***Joint Embedding*** model implementations (and also the ***Proposed Models*** that will be defined in next chapter) are trained and evaluated following the procedures depicted by algorithms 34 (**Train network function**) and 35 (**Evaluate network function**).

5.3.1 Training

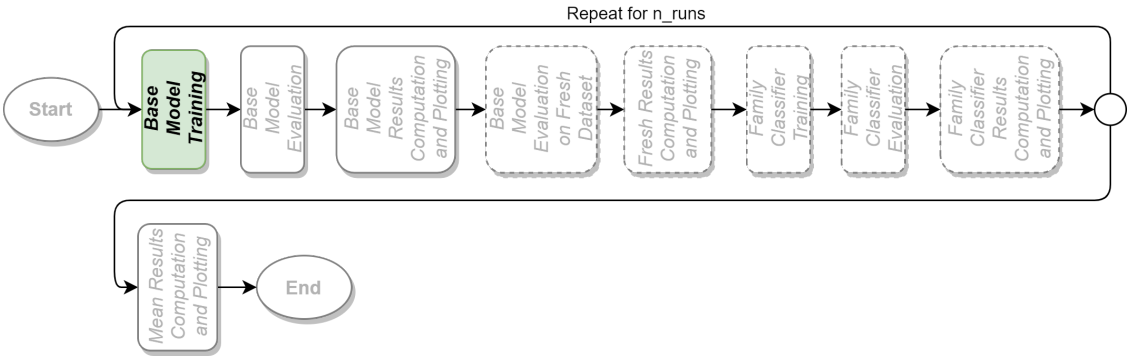


Figure 5.4: Model Training Workflow step

In particular, as shown in alg. 34, the training function first imports the chosen Net and Dataset generator (dataloder) implementations and some run variables situated in a *config* file, depending on the network and generator types selected by the user. Next, the model itself is defined calling the *init* member function of the previously imported *Net* class with all the needed parameters (such as the number of tags, the input feature dimension, the layer sizes, the dropout probability, etc.). Then, the selected optimizer (**Adam** or **SGD**) is instantiated by passing it the model parameters to optimize, and the chosen learning rate, weight decay and momentum. This is followed by the definition of the training and validation generators with the proper parameters. Next, the model is trained for '*epochs*' epochs. At each epoch the model is first set in training mode, then, for all the mini-batches randomly sampled from the training generator the optimizer gradients are re-set and the input features are forwarded through the network to get the output predictions. The loss for the current mini batch is then computed and logged. Finally the model parameters are updated by computing the loss gradients and back-propagating them through the network. Moreover, after each training epoch the model is also validated on samples provided by the validation generator. In particular, the model is first set in *evaluation* mode, then, for all the mini-batches of data got from the validation generator the input features are passed through the network to produce the output predictions which are in turn used to compute the loss term. The aggregate loss, in this case, is simply logged (no gradients are computed). Finally, at the end of each epoch the model and optimizer state parameters are saved to file.

5.3.2 Evaluation

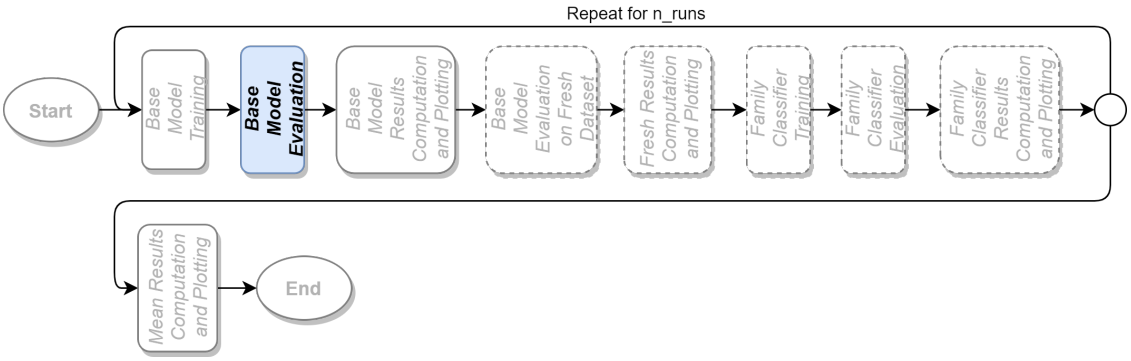


Figure 5.5: Model Evaluation Workflow step

The evaluation function, presented in 35, begins in a similar way as the training function by importing the Net and Dataset generator classes along with some run parameters. Then, the function continues by calling the model (*init*) member function with all the needed parameters

Algorithm 34 Train Network function

```
1: function TrainNetwork(ds_path, net_type, gen_type, batch_size, epochs,
   use_malicious_labels, use_count_labels, use_tag_labels, feature_dimension, workers, ...)
2:
3:     Net, Dataset, get_generator, run_params ← import_modules(net_type, gen_type)
4:
5:     model ← Net(use_malicious_labels, use_count_labels,
6:                 use_tag_labels, len(Dataset.tags),
7:                 feature_dimension, run_params['layer_sizes'],
8:                 run_params['dropout_p'], run_params['activation_function'],
9:                 run_params['normalization_function'])
10:
11:    opt ← run_params['optimizer'](model.parameters(), run_params['lr'],
12:                                  run_params['weight_decay'], run_params['momentum'])
13:
14:    generator ← get_generator(ds_path, batch_size,
15:                              'train', workers,
16:                              training_n_samples, use_malicious_labels,
17:                              use_count_labels, use_tag_labels)
18:
19:    val_generator ← get_generator(ds_path, batch_size,
20:                                  'validation', workers,
21:                                  validation_n_samples, use_malicious_labels,
22:                                  use_count_labels, use_tag_labels)
23:    ...
24:    for all epoch ∈ range(epochs) do
25:        model.train()
26:
27:        for all i, (features, labels) ∈ enumerate(generator) do
28:            opt.zero_grad()
29:            out ← model(features)
30:            loss_dict ← model.ComputeLoss(out, labels, run_params['loss_wts'])
31:            loss ← loss_dict['total']
32:            log(loss_dict)
33:            loss.backward()
34:            opt.step()
35:        end for
36:
37:        model.eval()
38:
39:        for all i, (features, labels) ∈ enumerate(val_generator) do
40:            torch.no_grad()
41:            out ← model(features)
42:            loss_dict ← model.ComputeLoss(out, labels)
43:            log(loss_dict)
44:        end for
45:
46:        model.save(epoch)
47:        save_opt_state(opt, epoch)
48:    end for
49: end function
```

and by loading the model state checkpoint chosen by the user. This function, however, then sets the model in *evaluation* mode and defines the test dataset generator. Then, for all the mini-batches of data got from the test generator the samples' input features are passed to the network and the corresponding predictions are produced and saved to file.

Algorithm 35 Evaluate Network function

```
1: function EvaluateNetwork(ds_path, checkpoint_file, net_type, gen_type, batch_size, evaluate_malware, evaluate_count, evaluate_tags, feature_dimension, ...)
2:
3:     Net, Dataset, get_generator, run_params ← import_modules(net_type, gen_type)
4:
5:     model ← Net(evaluate_malware, evaluate_count,
6:                 evaluate_tags, len(Dataset.tags),
7:                 feature_dimension, run_params['layer_sizes'],
8:                 run_params['dropout_p'], run_params['activation_function'],
9:                 run_params['normalization_function'])
10:
11:    model.load_state_dict(checkpoint_file)
12:    model.eval()
13:
14:    generator ← get_generator(ds_path, batch_size,
15:                              'test', workers,
16:                              test_n_samples, use_malicious_labels,
17:                              use_count_labels, use_tag_labels,
18:                              return_shas=True)
19:    ...
20:    for all shas, features, labels ∈ generator do
21:        predictions ← model(features)
22:        save_to_file(labels, predictions)
23:    end for
24: end function
```

5.3.3 Results Computation and plotting

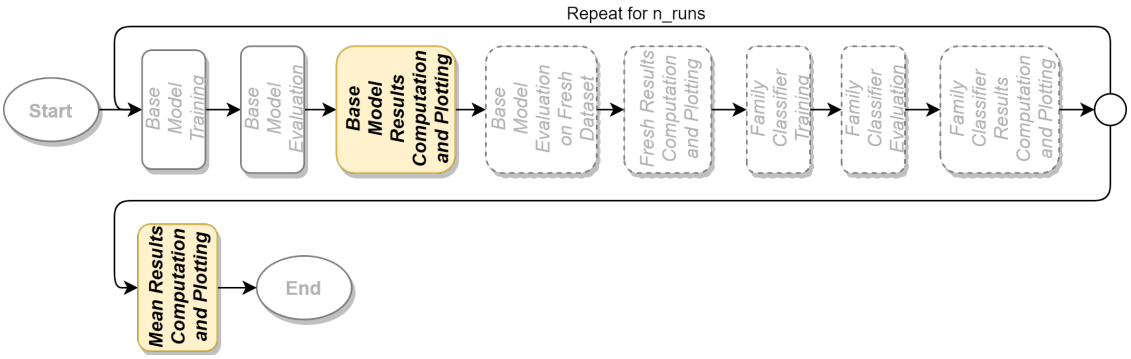


Figure 5.6: Results Computation and Plotting Workflow step

The actual binary cross entropy statistics (such as *TPR*, *Accuracy*, *Recall*, etc.. at different FPRs) and the resulting AUC-ROC scores and ROC curves are computed, plotted and saved to file by a distinct set of functions which are not presented in this document given that they simply implement and apply the corresponding mathematical formulas.

Chapter 6

Proposed Models

6.1 Malware Detection and Description via SMART tags Model

ALOHA model (described in section 5.1.1) is based on the idea of enhancing the model performance and generalization on the task of *Malicious/Benign* label prediction with respect to a more simple single-task model by exploiting additional tasks (*Vendor Count* prediction, *SMART Tags* prediction, etc.) which are jointly optimized in a Multi-objective/multi-task learning fashion. Multi-objective learning has, in fact, proven to be beneficial for the model generalization since it provides a regularization effect which results in overall better training/validation loss curves and arguably better evaluation results.

On the other hand, the **Joint Embedding** model (presented in section 5.1.2) was designed with the idea of constructing an embedding space in which both the samples and tag labels can be mapped to, such that each sample ends up close to the tags it is associated to (and possibly far from the other unrelated tags). The *Joint Embedding* model is trained by optimizing its parameters on the single task of tag prediction by means of tags' and samples' embeddings similarity, with no additional tasks.

The natural follow-up to those models is a fusion between their two base ideas ultimately creating a model with the main objective of creating a joint embedding space, as done by the **Joint Embedding** model, while exploiting Multi-objective learning during model training, as done by the **ALOHA** model. This model, which is called **Proposed Model** throughout this document, could potentially perform better in the SMART tag prediction task than the **Joint Embedding** model implementation given that during the training procedure its parameters are updated considering also the additional targets. Moreover, it could also perform better than the **ALOHA** model implementation in the *Malicious/Benign* label prediction task since the SMART tag prediction task, which is solved by the proposed model by learning the joint embedding and calculating similarities, is harder than the original solution based on multi-label binary classification with one head for each tag. Harder complementary tasks are, in fact, often used when exploiting *multi-task* learning because they help in finding a more informed latent representation of samples which in turn leads to better performances during testing/deployment.

Moreover, the fact that the learned embedding should pull samples close to their associated tags (and far from the others) also means that samples which are similar or exhibit similar capabilities (sharing the same tags) should be close to each other in the latent space. In turn, the representation of PE files in the learned embedding space, along with the distance function used in this space, enables the computation of file-to-file similarities. This makes it possible to measure how similar two malware samples are in terms of their capabilities. The **Joint Embedding** and **Proposed Model** approaches are therefore interesting from the perspective of predicting the specific malware family PE files belong to based on their distances with respect to known malware samples. Notice, however, that both the **Joint Embedding** and the **Proposed Model** network are trained with a different objective - that of predicting the SMART tags associated

to each malware sample (and also the *malicious/benign* label and *vendor count* for the proposed model). It is not uncommon in the ML field to train a model on a specific task and then testing (or deploying) it on a different, but most probably related, task with little degradation in model performance. Moreover, the models are trained on a dataset of malware samples which intrinsically belong to different malware families (although the effective family labels are not provided in the dataset), therefore the information of each sample's family is implicitly present.

Therefore the **Proposed Model** and the **Joint Embedding** model implementation performances are not only evaluated on the tasks they ware trained for (*Malicious/Benign* label and *Vendor Count* prediction tasks and/or *SMART tag* prediction task), but also on the **f-way Malware Family Classification** task and on the **Malware Family Ranking** task. Again, for the reasons mentioned above, the **Proposed Model** should perform better than the **Joint Embedding** model implementation also in these last tasks.

If the **Proposed Model** performs well enough on the malware family *prediction* and *ranking* tasks, it could be possible to use its learned embedding space to define implicit signatures - the corresponding representation in this space, a.k.a. the embedding - of malware samples belonging to a specific malware family. These signatures could then be used to craft Yara Rules - with the use of a specially crafted custom Yara module that dynamically extracts the embedding representation for new PE files and compares them to the known signatures - to detect similar samples as belonging to the specific family represented by each rule. Yara Rules created in this way could potentially inherit the good properties of ML methods such as being able to detect not only known malware strains but also novel variants, being less susceptible to obfuscation attempts, etc. while being compatible with the highly expressive approach that is Yara.

6.1.1 Implementation

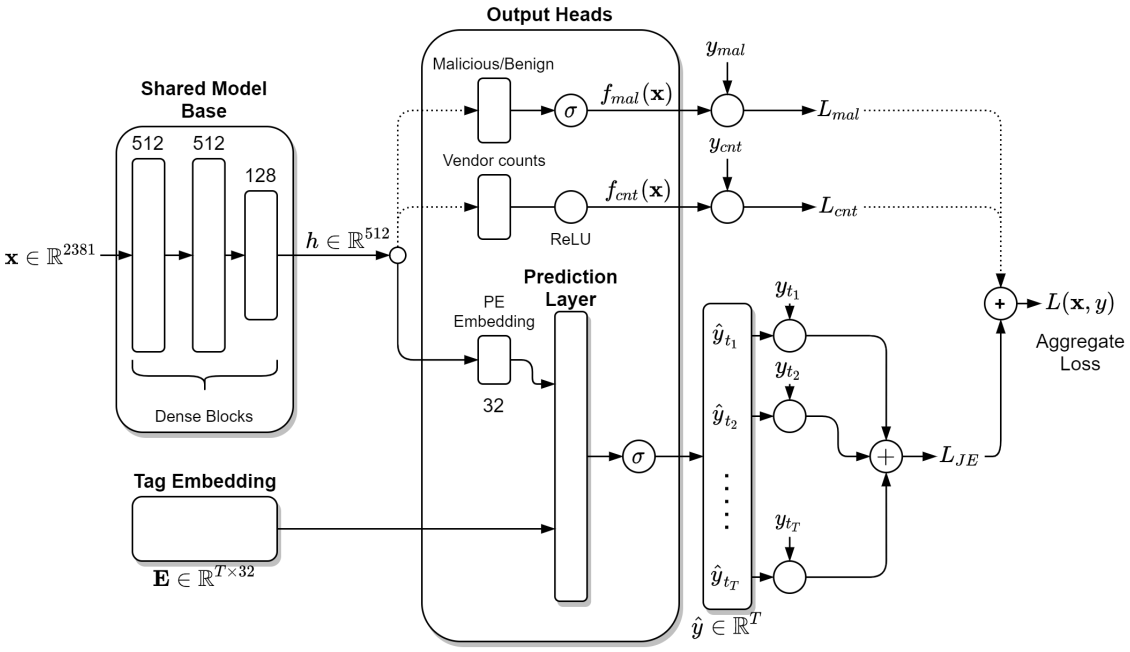


Figure 6.1: Proposed Model architecture

The proposed model implementation (fig. 6.1) consists of 3 main topologies: the *Shared Model Base* sub-network, the *Tag Embedding* and the multiple parallel *Output Heads*.

The model *shared base* topology is composed of 5 sequential dense blocks of output sizes 1024, 768, 512, 512 and 512 respectively. These dense blocks have the same form as those used in the *ALOHA* and *Joint Embedding* model implementations described above (in 5.1.1 and 5.1.2 respectively): in particular a linear layer with dropout is used, followed by batch normalization

and ReLU activation function. Again at network definition it is possible to dynamically set the parameters for this base topology such as the exact number of layers, the linear layer's sizes, the dropout probability p_d, the normalization function and the non-linear activation function used. This *shared base* topology transforms a given input sample feature vector $\mathbf{x}$ (of size 2381) into its intermediate representation h of size 512 which is then used as input to the parallel output heads.

The *Tag Embedding* matrix $\mathbf{E} \in \mathbb{R}^{T \times 32}$, which maps the tags to their corresponding representation in the Joint Embedding space, is the same as the one used in the *Joint Embedding* model implementation (5.1.2) and is therefore also equal to the one used in the original implementation described in [3]. This matrix is used as second input for the *Prediction Layer* of the tag output head.

Similarly to the *ALOHA* model implementation (5.1.1), this model uses multiple parallel *output heads*, each relative to a different task, whose values are jointly optimized during training by back-propagating the gradients computed from an aggregate loss. In particular, the tasks used in this case are: the *Malicious/Benign label* prediction task (optional), the *Vendor Count* estimation task (optional) and the *SMART Tag* prediction task based on the similarity between samples' and tags' embeddings. The first two output heads (the *Malicious/Benign label* prediction head and the *Vendor Count* estimation head) are the same ones used in the *ALOHA* model implementation (5.1.1) and can optionally be turned on or off at network definition.

In particular, the *Malicious/Benign* label prediction head is composed of a single dense layer followed by the *sigmoid* non-linearity. The loss between the output of the malware/benign head $f_{mal}(x)$ and the ground truth label y_{mal} is computed using the binary cross-entropy loss function.

The *Vendor Count* prediction head, instead, consists of a single linear layer, followed by the ReLU non-linearity, whose parameters are updated in order to optimize the negative log-likelihood loss, as described in 3.6.1 (and [2]), between the predicted and the ground truth count value.

The *Tag* prediction head, on the other hand, is always on and consists of the same components used in the *Joint Embedding* model implementation (5.1.2): the linear layer which maps each sample intermediate representation h to its *PE Embedding* in the **Joint Embedding space** and the *Prediction Layer*, which produces a similarity score matrix between samples' and tags' embeddings. The resulting matrix is then run through a *sigmoid* non-linearity (by value) to produce the tags probabilities for sample $\mathbf{x}$. The similarity function used in this model can be chosen to be one of the following at network definition: *dot product* (5.1), *cosine similarity* (5.2) or one of the *inverse Euclidean distances* (*exp* (5.3), *inv* (5.4) or *inv_pow* (5.5)). The loss for the *Tag prediction head* is computed as the sum over all tags of the individual cross entropy losses between the predicted tags and the ground truth labels, as described in 3.6.2 (and [3]).

Finally the individual head losses are aggregated together by computing their weighted sum. Different weights can be assigned to the distinct heads in order to tune their importance in the final loss term, in turn affecting how the model's parameters are updated during back-propagation.

Net Definition

As presented by algorithm 36 (***Proposed Model* init function**), the ***Proposed Model*** is initialized first by defining the *model shared topology* (*model_base*) as a sequence of blocks composed each of a linear layer, a Normalization layer, an activation function and dropout, exactly as previously done for the ***ALOHA*** and ***Joint Embedding*** model implementations. Next, similarly to the ***Joint Embedding*** implementation, the *pe_embedding* topology and the *tags_embedding* matrix are defined: the first as a single linear layer with output size equal to the *embedding_dimension* (= 32), followed by a Normalization layer and an activation function, and the latter as an embedding matrix of size (*n_tags*, *embedding_dimension*). The most important novelty compared to the ***Joint Embedding*** model is the presence of also the *Malicious/Benign* label and *Vendor Count* prediction heads from the ***ALOHA*** model implementation. In particular, the *Malicious/Benign* (*malware_head*) is defined as a linear layer of output size 1 immediately followed by a *sigmoid* non linearity. The *Vendor Count* head (*count_head*), on the other hand, is set to be a simple linear layer trailed by the *ReLU* non-linearity.

Algorithm 36 Proposed Model Net class, Init

```
1: class NET
2:     function INIT(self, use_malware, use_counts, use_tags, n_tags, feature_dimension, embedding_dimension, max_embedding_norm, layer_sizes, dropout_p, activation_function, normalization_function)
3:
4:         self.use_malware ← use_malware
5:         self.use_counts ← use_counts
6:         self.n_tags ← n_tags
7:         self.embedding_dimension ← embedding_dimension
8:         layers ← [ ]
9:
10:        if layer_sizes is None then
11:            layer_sizes ← [512, 512, 128]
12:        end if
13:
14:        for all i, ls ∈ enumerate(layer_sizes) do
15:            if i = 0 then
16:                layers.append(nn.Linear(feature_dimension, ls))
17:            else
18:                layers.append(nn.Linear(layer_sizes[i −1], ls))
19:            end if
20:
21:            layers.append(normalization_function(ls))
22:            layers.append(activation_function())
23:            layers.append(nn.Dropout(dropout_p))
24:        end for
25:
26:        self.model_base ← nn.Sequential(*tuple(layers))
27:        self.sigmoid ← nn.Sigmoid()
28:
29:        self.pe_embedding ← nn.Sequential(
30:            nn.Linear(layer_sizes[-1], self.embedding_dimension),
31:            normalization_function(self.embedding_dimension),
32:            activation_function())
33:
34:        self.malware_head ← nn.Sequential(nn.Linear(layer_sizes[-1], 1),
35:                                          nn.Sigmoid())
36:
37:        self.count_head ← nn.Sequential(nn.Linear(layer_sizes[-1], 1), nn.ReLU())
38:
39:        self.tags_embedding ← nn.Embedding(self.n_tags, self.embedding_dimension,
40:                                           max_norm=max_embedding_norm)
41:    end function
```

The input data (PE files' numerical feature vectors) is forwarded through the model as shown in alg. 37 (***Proposed Model* forward function (with *dot product*)**). In particular, it first passes through the model base topology (*model_base*) to produce an intermediate representation *base_out* which is then forwarded to the various parallel output heads. The *malware label prediction head* and the *vendor count prediction head*, if enabled, produce the corresponding predictions given as input the transient latent representation *base_out*. On the other hand, for the *tags prediction* topology, the intermediate representation *base_out* is fed into the *pe_embedding* sub-net producing the latent representation of the input data samples in the joint embedding space (PE embedding). The tag similarity scores are then computed by applying the dot product (via a simple matrix multiplication) between the PE embedding and the tags embedding matrix (*tags_embedding*). Finally, the tags probability predictions are produced by applying the *sigmoid* activation function

Algorithm 37 Proposed Model Net class, Forward (with *Dot Product*)

```
42:  function FORWARD(self, data)
43:      rv ← {}
44:      base_out ← self.model_base(data)
45:
46:      if self.use_malware then
47:          rv['malware'] ← self.malware_head(base_out)
48:      end if
49:
50:      if self.use_counts then
51:          rv['count'] ← self.count_head(base_out)
52:      end if
53:
54:      pe_embedding ← self.pe_embedding(base_out)
55:      tags_embedding ← self.tags_embedding((Dataset.encoded_tags))
56:      rv['similarity'] ← torch.matmul(pe_embedding, tags_embedding.T)
57:      rv['probability'] ← self.sigmoid(similarity_scores)
58:
59:      return rv
60:  end function
```

Algorithm 38 Proposed Model Net class, Forward (with *Cosine similarity*)

```
61:  function FORWARD(self, data)
62:      ...
63:      similarity_scores ← torch.div(
64:          torch.add(Cosine_similarity(pe_embedding, tags_embedding, dim=2), 1.0), 2.0)
65:      rv['similarity'] ← similarity_scores
66:      rv['probability'] ← similarity_scores
67:      return rv
68:  end function
```

Algorithm 39 Proposed Model Net class, Forward (with *Inverted Pairwise Euclidean Distance*)

```
69:  function FORWARD(self, data)
70:      ...
71:      distances ← torch.cdist(pe_embedding, tags_embedding, p=2.0)
72:      similarity_scores ← DistanceToSimilarity(distances, a=sim_function_a,
73:                                               function=sim_function)
74:      rv['similarity'] ← similarity_scores
75:      rv['probability'] ← similarity_scores
76:      return rv
77:  end function
```

on the previously computed similarity scores.

If, however, the user chooses to try a different similarity function other than the ***dot product***, two alternatives are available: the ***cosine similarity*** or the ***inverted pairwise euclidean distance***. In particular both the ***Proposed Model* forward function (with *cosine similarity*)**, shown in alg. 38, and the ***Proposed Model* forward function (with *inverted pairwise Euclidean distance*)**, shown in alg. 39, are for the most part identical to the original forward function but for the computation of the similarity scores. The *cosine similarity* version of this function, in fact, computes the similarity scores as the cosine similarities between the PE files' and tags' embeddings, which are values in the range [−1,1], normalized to the [0,1] interval. On the other hand, the *inverted pairwise Euclidean distance* version computes those scores as the euclidean distances between the two embeddings, inverted through the ***DistanceToSimilarity*** function already mentioned in sec. 5.1.2. Again, similarly to what was done in the *Joint Embedding* model implementation, in both cases the final tag probabilities coincide with the computed

Algorithm 40 Proposed Model Net class, Compute Loss

```
78:   function ComputeLoss(predictions, labels, loss_wts)
79:       if loss_wts is None then
80:           loss_wts ← {'malware': 1.0, 'count': 0.1, 'tags': 1.0}
81:       end if
82:       loss_dict ← {'total': 0.}
83:
84:       if 'malware' ∈ labels then
85:           malware_loss ← Binary_cross_entropy(predictions['malware'], labels['malware'])
86:           if 'malware' ∈ loss_wts then
87:               weight ← loss_wts['malware']
88:           else
89:               weight ← 1.0
90:           end if
91:
92:           loss_dict['malware'] ← malware_loss
93:           loss_dict['total'] ← loss_dict['total'] + malware_loss × weight
94:       end if
95:
96:       if 'count' ∈ labels then
97:           count_loss ← PoissonNLLLoss(predictions['count'], labels['count'])
98:           if 'count' ∈ loss_wts then
99:               weight ← loss_wts['count']
100:          else
101:              weight ← 1.0
102:          end if
103:
104:          loss_dict['count'] ← count_loss
105:          loss_dict['total'] ← loss_dict['total'] + count_loss × weight
106:      end if
107:
108:      if 'tags' ∈ labels then
109:          similarity_loss ← Binary_cross_entropy(predictions['probability'], labels['tags'],
110:              reduction='none').sum(dim=1).mean(dim=0)
111:          if 'tags' ∈ loss_wts then
112:              weight ← loss_wts['tags']
113:          else
114:              weight ← 1.0
115:          end if
116:
117:          loss_dict['jointEmbedding'] ← similarity_loss
118:          loss_dict['total'] ← loss_dict['total'] + similarity_loss × weight
119:      end if
120:
121:      return loss_dict
122:  end function
123: end class
```

similarity scores.

The ***Proposed Model*** implementation uses exactly the same ***GetEmbedding*** and ***GetSimilarity*** functions as the ones used in the ***Joint Embedding*** model implementation, shown in alg. 28. Again, these functions are particularly useful when performing evaluations on the learned embedding.

Finally, algorithm 40 (***Proposed Model* compute loss function**) describes how the ***Proposed Model*** loss computation is implemented in *Python*/*Pytorch* code. In particular the final total loss corresponds to the weighted sum of the losses from each enabled output head. The

weights used in this computation can be defined by the user at network definition. Both the *Malicious/Benign* label prediction loss and the *Vendor Count* prediction loss computations were taken from the ***ALOHA*** model implementation. Therefore the first is implemented as the **binary cross entropy loss** between the predicted malware label and the respective ground truth label, the latter instead corresponds to the **Negative Log Likelihood with Poisson distribution** between the predicted count and the ground truth value. On the other hand, the *Tags* prediction loss computation is taken from the ***Joint Embedding*** model implementation and is therefore the per-sample mean of all tag losses, each of which is computed as the **binary cross entropy** loss between the predicted tag probability and the ground truth tag.

6.1.2 Experiments

The proposed model was trained multiple times for 10 *epochs* on the first $6M$ training and 1.153.846 validation samples of the pre-processed Sorel20M dataset (loaded with generator ***alt3***) with both the optional output heads (*Malicious/Benign* label prediction and *Vendor Count* prediction) on, changing the network hyper-parameters like the number of layers, layers dimensions (number of hidden units per layer), normalization function, activation function and dropout value for the base topology. Other hyper-parameters that were iteratively changed were the used optimizer (**Adam** or **SGD**), learning rate (LR), momentum, weight decay, loss weights and the similarity score function used (chosen between the *dot product*, the *cosine similarity* and one of the *inverted Euclidean Distances*: *exp*, *inv* or *inv_pow*).

Then, for the evaluation of the resulting model the first 1.846.154 test samples of the Sorel20M pre-processed dataset were used to asses the model *TPR*, *accuracy*, *recall*, *precision*, *f1 score*, *jaccard similarity*, and *mean-per-sample accuracy* at different *FPRs* of interest (particularly at $FPR = 1\%$) for the tags and malicious/benign label prediction task, as well as the corresponding ROC curves and AUC scores.

Moreover, this model learned samples' representation (embedding), together with the *Joint Embedding* model implementation one, was also tested on the fresh dataset, which contains 10.000 malicious samples belonging to 10 different malware families to asses its performance in the malware family classification and ranking tasks with respect to the mentioned previous method. The malware families represented by the fresh dataset were chosen among the most wide-spread PE malware families present in Italy at the time of writing (as reported by [53]) for which *Malware Bazaar* [54] provided at least 1.000 samples.

In particular, the learned representation was first evaluated on the task of *f-way* family classification via nearest neighbour search as previously described in 4.2.2 by randomly sampling k ($\in [1, 10]$) anchor samples and q ($= 23$) query samples, per-family, and assigning to the ($f \times q = 230$) query samples the same class as their closest anchor sample in feature space. The sampling process (for both anchors and query samples), classification and evaluation was repeated 15 times per value of k to obtain uncertainty estimates for the results. The trends of some binary cross entropy statistics like *accuracy*, *precision*, *recall* and *f1-score* were computed along with the ROC-AUC score trends and some confusion matrixes.

Then, the learned embedding was evaluated on the malware family ranking task (as described in 4.2.2) by randomly selecting $\boldsymbol{q}$ (100) query samples from the fresh dataset and computing, for each of them, the ranking of the other $n_{samples} - 1 = 9.999$, ordering them by similarity. Then, given the resulting rankings, the **MAP** and **MRR** scores were calculated.

The final **Proposed Model** architecture consisted of 5 layers of sizes 1024, 768, 512, 512, 512 respectively (keeping them consistent to the *ALOHA* and *Joint Embedding* model implementation in order to be able to compare their results), with dropout probability $p_d = 5\%$, *Batch* normalization, **ReLU** activation function and the *dot product* as similarity score function. The training was carried out with a learning rate $LR = 10^{-3}$, **Adam** optimizer, no momentum nor weight decay, and loss weights set to 1.0 for the malicious/benign head, 0.1 for the count head and 1.0 for the SMART tag (joint embedding) head.

The **Proposed Model** evaluation results are reported in chapter 7.

6.1.3 Training and Evaluation algorithms

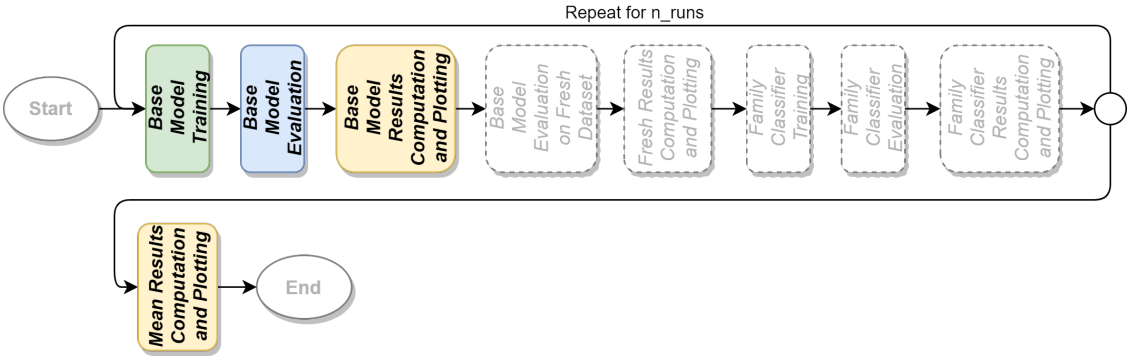


Figure 6.2: Model Training and Evaluation Workflow steps

As previously mentioned in section 5.3, the ***Proposed Model*** training and evaluation procedures are the same as the ones used for the ***ALOHA*** and ***Joint Embedding*** model implementations. In particular, the training procedure is presented in alg. 34, while the model evaluation is shown in alg. 35.

6.1.4 Evaluate Fresh algorithm

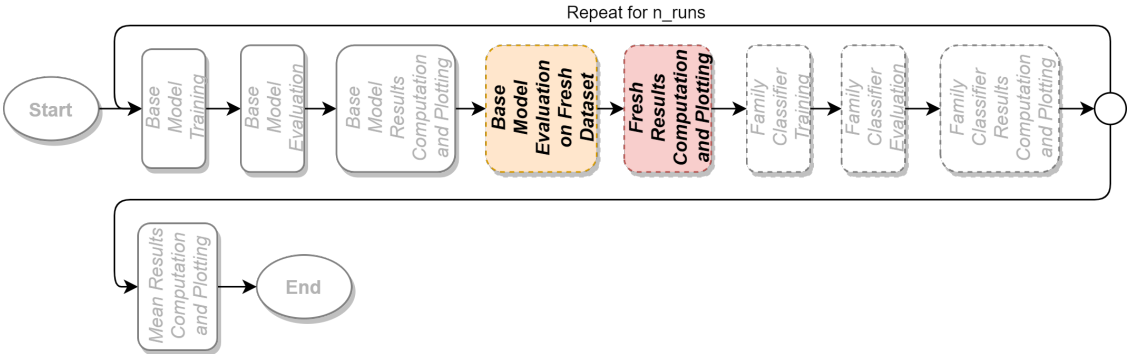


Figure 6.3: Model Fresh Dataset Evaluation Workflow steps

Algorithm 41 Evaluate Fresh function

```
1: function EvaluateFresh(fresh_ds_path, checkpoint_path, net_type, min_n_anchor_samples,
   max_n_anchor_samples, n_query_samples, n_evaluations, batch_size)
2:
3:     EvaluateFreshScores(fresh_ds_path, checkpoint_path, net_type,
4:                         n_query_samples, min_n_anchor_samples,
5:                         max_n_anchor_samples, n_evaluations, batch_size)
6:
7:     EvaluateFreshRankings(fresh_ds_path, checkpoint_path, net_type,
8:                           n_query_samples, n_evaluations, batch_size)
9: end function
```

As previously mentioned, both the ***Proposed Model*** and the ***Joint Embedding*** model implementations are also evaluated on the *f-way* malware family classification and family ranking tasks. In particular the function presented in algorithm 41 (**Evaluate fresh function**) is used to produce both the mentioned evaluation results. More specifically, it calls the following two functions in succession: ***EvaluateFreshScores*** and ***EvaluateFreshScores***.

Model Evaluation on f-way Malware Family Classification/Family Ranking tasks

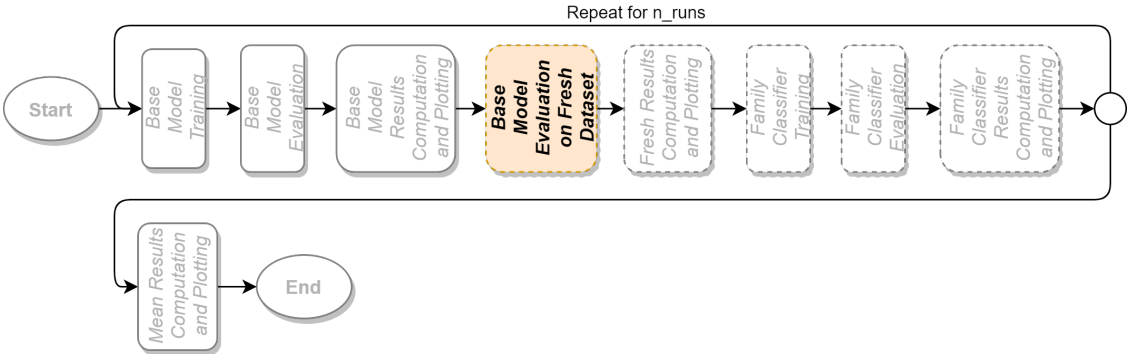


Figure 6.4: Model Fresh Dataset Evaluation Workflow step

Algorithm 42 Evaluate Fresh Scores function

```
function EvaluateFreshScores(ds_path, checkpoint_path, net_type, n_queries,
min_anchors, max_anchors, n_evaluations, batch_size)

    Net, run_params ← import_modules(net_type)

    model ← Net(use_malware=False, use_counts=False, n_tags=len(Dataset.tags),
                feature_dimension=2381, embedding_dimension=32,
                layer_sizes=run_params['layer_sizes'],
                dropout_p=run_params['dropout_p'],
                activation_function=run_params['activation_function'],
                normalization_function=run_params['normalization_function'])

    model.load_state_dict(checkpoint_path)
    model.eval()
    generator ← get_generator(ds_path, batch_size, return_shas=True, shuffle=True)
    predictions ← {}

    for all k, n_anchors ∈ enumerate(range(min_anchors, max_anchors +1)) do
        predictions[n_anchors] ← [ ]

        for all j ∈ range(n_evaluations) do
            anchors ← GetSamples(model, generator, n_families, n_anchors)
            queries ← GetSamples(model, generator, n_families, n_queries, other=anchors)
            similarity_scores ←model.GetSimilarity(queries['embeddings'],
                                                    anchors['embeddings'])['similarity']
            predictions[n_anchors].append({
                'families': [label_to_sig(lab) ∀ lab ∈ range(n_families)],
                'shas': queries['shas'],
                'labels': queries['labels'],
                'predictions': anchors['lables'][argmax(similarity_scores)],
                'probabilities': nn.Softmax([[max(sims[[j ∀ j ∈ range(len(sims))
                    if anchors['labels'][j] = i]]) ∀ i ∈ range(n_families)]
                    ∀ sims ∈ similarity_scores])})
        end for
    end for

    save_to_file(predictions)
end function
```

Algorithm 43 Evaluate Fresh Rankings function

```
function EvaluateFreshRankings(ds_path, checkpoint_path, net_type, n_query_samples,
n_evaluations, batch_size)

    Net, run_params ← import_modules(net_type)

    model ← Net(use_malware=False, use_counts=False, n_tags=len(Dataset.tags),
                feature_dimension=2381, embedding_dimension=32,
                layer_sizes=run_params['layer_sizes'],
                dropout_p=run_params['dropout_p'],
                activation_function=run_params['activation_function'],
                normalization_function=run_params['normalization_function'])

    model.load_state_dict(checkpoint_path)
    model.eval()
    generator ← get_generator(ds_path, batch_size, return_shas=True, shuffle=True)
    ranking_scores ← 'MRR': [ ], 'MAP': [ ]
    global_ranks_to_save ← None

    for all j ∈ range(n_evaluations) do
        queries ← GetSamples(model, generator, n_families, n_query_samples)
        rank_per_query ← [ ]

        for all shas, features, labels ∈ generator do
            embeddings ← model.GetEmbedding(features)émbedding']
            similarity_scores ← model.GetSimilarity(queries['embeddings'],
                                                    embeddings)['similarity']

            for all i, s ∈ enumerate(queries['shas']) do
                indices ← -similarity_scores[i, [j ∀ j ∈ range(len(similarity_scores[i]))
                                                 if shas[j] ≠ s]].argsort()
                rank_per_query.append({
                    'query_sha': s,
                    'ground_truth_label': queries['labels'][i],
                    'ground_truth_family': label_to_sig(queries['labels'][i]),
                    'rank_shas': shas[indices],
                    'rank_labels': labels[indices],
                    'rank_families': [label_to_sig(lab) ∀ lab ∈ labels[indices]]
                })
            end for
        end for
        ranking_scores, global_ranks_to_save, rank_per_query ← ComputeRankingScores(
            ranking_scores, global_ranks_to_save, rank_per_query)
    end for

    save_to_file(global_ranks_to_save)
end function
```

The ***EvaluateFreshScores*** function, shown in alg. 42, begins by importing the '*Net*' class corresponding to the chosen *net_type* and then instantiates it by providing the needed arguments (such as the *feature_dimension*, the *embedding_dimension*, the *layer_sizes*, etc.). Next, the function loads the model checkpoint state parameters corresponding to the selected training run, it sets the model into *evaluation* mode and instantiates the fresh dataset generator (dataloader). After these preliminary steps, for all the values of '*n_anchors*' between '*min_anchors*' and '*max_anchors*' (included), the function performs '*n_evaluations*' evaluations recording the model predictions, which will be later used to compute the corresponding accuracy (and other statistics) results. In

Algorithm 44 Compute Ranking Scores function

```
1: function ComputeRankingScores(ranking_scores, global_ranks_to_save, rank_per_query)
2:     rs ← binarize(rank_per_query)
3:     log(mean_reciprocal_rank(rs))
4:     log(mean_average_precision(rs))
5:
6:     queries_indexes ← {
7:         'max_rr': max_reciprocal_rank_index(rs),
8:         'min_rr': min_reciprocal_rank_index(rs),
9:         'max_ap': max_average_precision_index(rs),
10:        'min_ap': min_average_precision_index(rs)
11:    }
12:
13:    ranks_to_save ← {key: rank_per_query[index] ∀ key, index ∈ queries_indexes.items()}
14:    return ranking_scores, global_ranks_to_save, rank_per_query
15: end function
```

particular, for each evaluation '*n_anchors*' anchor samples and '*n_queries*' query samples, per-family, are randomly selected (without overlaps) from the fresh dataset using the **GetSamples** function and the similarity scores between their embeddings are computed by using the model's **GetSimilarity** function. Finally, the function produces and saves to file the family predictions, by getting, for each query sample, the family label of the most similar anchor sample in the previously computed similarity score matrix. Moreover, the function also computes the per-sample family label probabilities by retrieving the similarity score corresponding to the most similar anchor sample, for each family, and subsequently applying the *Softmax* function to the resulting list of family scores.

The ***EvaluateFreshRankings*** function, presented in algorithm 43, again begins by importing the selected *Net* class. It then instantiate the model with the needed parameters, loads the selected checkpoint state parameters, sets the model to *evaluation* mode and defines the fresh dataset generator (dataloader). Next, for '*n_evaluations*' times, the function randomly selects '*n_query_samples*' query samples, per family, from the previously defined fresh dataset *generator*. The function then cycles through all the mini-batches of data from the fresh dataset generator and gets the samples embeddings by feeding the model with their numerical feature vectors. Then, in order to generate the family rankings, the function computes the similarity scores between the **query** samples' embeddings and the mini-batch embeddings (which may contain also the **query** sample latent representation) by using the model's *GetSimilarity* function. Next, for each **query** sample the function generates one ranking by ordering the samples' *shas*, *family labels* and *family names* by their similarity with the current **query** sample (excluding the current **query** sample from the ranking). Finally, the function uses the **ComputeRankingScores** function, presented in alg. 44, to compute the *MRR* and *MAP* scores and save a bunch of interesting rankings, namely the ones which produced the maximum and minimum RR (reciprocal rank) and AP (average precision), for the current evaluation. The overall mean and standard deviation of both ranking scores are computed and saved at the end of the cycle.

The **GetSamples** function (alg. 45), used by functions *EvaluateFreshScores* and *EvaluateFreshRankings* and presented in alg. 45 selects '*n_samples_to_get*' samples for each of the '*n_families*' families from the provided *generator* (fresh dataset) excluding samples present in the '*other*' set (if provided). In order to do this, the function cycles through the provided *generator* and gets the *shas*, *labels*, *features* and *embedding* of the samples not in the '*other*' set and then selects, for each family, as many samples as possible until all families have exactly '*n_samples_to_get*' samples.

Algorithm 45 Get Samples function

```
function GetSamples(model, generator, n_families, n_samples_to_get, other)
    samples ← None
    samples_families ← [0 ∀ _ ∈ range(n_families)]

    for all shas, features, labels ∈ generator do
        if other is not None then
            indices ← [i ∀ i, sha ∈ enumerate(shas) if sha not in other['shas']]
            shas ← [shas[i] ∀ i ∈ indices]
            features ← features[indices]
            labels ← labels[indices]
        end if

        embeddings ← model.GetEmbedding(features)['embedding']
        for all n ∈ range(n_families) do
            if samples_families[n] ≥ n_samples_to_get then continue
            end if

            indices ← [i ∀ i, label ∈ enumerate(labels) if label = n]
            indices ← indices[:n_samples_to_get] if len(indices) > n_samples_to_get else indices
            if samples is None then
                samples ← {'shas': [indices],
                           'labels': labels[indices],
                           'features': features[indices],
                           'embeddings': embeddings[indices]}
            else
                samples['shas'].extend(shas[indices])
                samples['labels'].extend(labels[indices])
                samples['features'].extend(features[indices])
                samples['embeddings'].extend(embeddings[indices])
            end if

            samples_families[n] ← samples_families[n] + len(indices)
        end for

        if all(n ge n_samples_to_get ∀ n ∈ samples_families) then break
        end if
    end for

    return samples
end function
```

Fresh Results Computation and Plotting

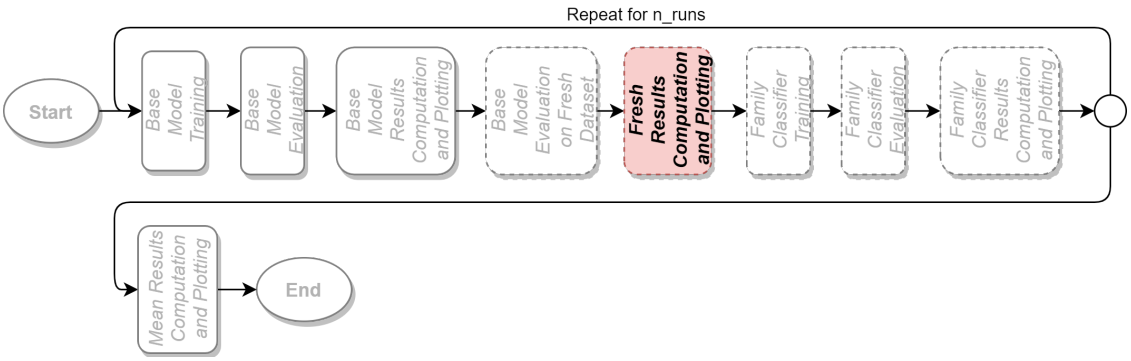


Figure 6.5: Fresh Results Computation and Plotting Workflow step

The actual binary cross entropy statistics trends for the *f-way* malware family classification task (*Accuracy*, *Recall*, *precision* etc..) and the resulting *AUC-ROC* score trends are computed, plotted and saved to file by a distinct set of functions which are not presented in this document given that they simply implement and apply the corresponding mathematical formulas.

6.2 Malware Family Classifier

Given the **Proposed Model** relatively poor results in the Malware Family Classification task (which it was never trained for), the next step was the development of a **Malware Family Classifier** specifically trained (and tested) on the fresh dataset (which contains information about the malware family each sample belongs to) for such task. However, this dataset, which contains 10.000 samples divided in 7.000 training, 1.000 validation and 2.000 test samples, is arguably too small to produce a general enough model able to reliably predict the family label for each malware sample without overfitting on the training sub-split. This problem was therefore tackled with the aid of **transfer learning**.

Transfer Learning (TL) is a machine learning (ML) technique, especially popular in deep learning, that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. Transfer Learning is usually defined in terms of domains and tasks. A domain $\mathcal{D}$ consists of: a feature space $\mathcal{X}$ and a marginal probability distribution $P(\mathcal{X})$, where $\mathcal{X} = x_1....,x_n \in \mathcal{X}$. Given a specific domain, $\mathcal{D} = \mathcal{X}, P(\mathcal{X})$, a task consists of two components: a label space $\mathcal{Y}$ and an objective predictive function $f : \mathcal{X} -> \mathcal{Y}$. The function f is used to predict the corresponding label $f(x)$ of a new instance x. This task, denoted by $\mathcal{T} = \mathcal{Y}, f(x)$, is learned from the training data consisting of pairs x_i, y_i, where $x_i \in \mathcal{X}$ and $y_i \in \mathcal{Y}$. Given a source domain $\mathcal{D}_S$ and learning task $\mathcal{T}_S$, a target domain $\mathcal{D}_T$ and learning task $\mathcal{T}_T$, where $\mathcal{D}_S \neq \mathcal{D}_T$, or $\mathcal{T}_S \neq \mathcal{T}_T$, transfer Learning aims to help improve the learning of the target predictive function $f_T(\cdot)$ in $\mathcal{D}_T$ using the knowledge in $\mathcal{D}_S$ and $\mathcal{T}_S$. In practice, when using transfer learning, a base network is first trained on a base (large) dataset solving a specific task, and then the learned features are repurposed, or transferred, to a second target network to be trained on a target dataset and task.

Transfer Learning is typically exploited in one of the following two ways:

- **Feature Extraction**. In this case, the latent representations learned by a previous network - trained on the source domain to solve the source task - are used to extract meaningful features from new samples. A new classifier is then simply added on top of the pre-trained model and trained from scratch on the target domain, to solve the target task. Given that the shared base network already contains features that are generically useful for classifying samples, it is not needed to re(train) the entire model.
- **Fine Tuning**. Here, a few of the last layers of a frozen pre-trained model base are unfreezed and jointly trained together with the newly-added classifier layers. The higher-order feature representations are thus 'fine-tuned' in the model base topology in order to make them more relevant for the specific target task.

The intuition behind using Transfer Learning in this context is the following: a model trained on a large and general enough dataset of PE samples (such as the **Proposed Model** trained on Sorel-20m dataset) could effectively serve as a generic model of the world of PE files. These learned feature representations can then be taken advantage of while solving the different but related task of Malware Family Classification without having to start from scratch by training a model on a large dataset. This in turn should make it possible to train a Malware Family Classifier using the relatively small Fresh Dataset while obtaining meaningful results.

A Malware Family Classifier was thus constructed by adding a multi-class classifier sub-network on top of the PE embedding part of the trained **Proposed Model** - thus exploiting the learned embedding representation of PE files. The resulting model was trained (and validated) on the training (and validation) sub-split of the fresh dataset such that the imported Proposed Model parameters are just 'fine-tuned' while the newly added classifier's ones are learned from scratch. This practice should provide improved results in the Malware Family Classification task with

respect to using just the Proposed Model learned similarity embedding (as presented in last section) as well as with respect to a similarly complex family classifier trained entirely from scratch with the 'small' fresh dataset.

In order to evaluate the effectiveness of this approach with respect to simply training an equivalent family classifier entirely from scratch, two models were defined and trained with the same model hyper-parameters (learning rate, number of layers, etc.) and for the same amount of epochs: one had its model base parameters imported from a trained *Proposed Model* and 'fine-tuned' while the other was trained entirely from scratch. The resulting two models were then evaluated by recording and comparing their accuracy and some '*micro*' (global average), '*macro*' (unweighted per-class average) and '*weighted*' (per-class average, weighted by support) multi-class classification statistics such as *Jaccard-score*, *recall*, *precision* and *f1-score* along with the corresponding *AUC-ROC* scores.

6.2.1 Implementation

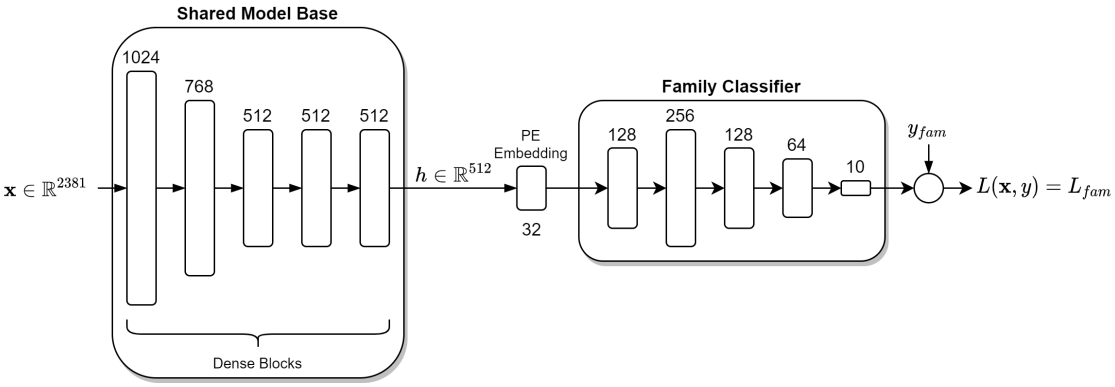


Figure 6.6: Family Classifier Model architecture

As shown in figure 6.6, the Malware Family Classifier model architecture consists of 3 main topologies: the *Shared Model Base* topology, the *PE Embedding* layer and the *Family Classifier* output head.

In particular, the *Shared Base* topology and the *PE Embedding* layer are defined to be equal to the ones used by the Proposed Model (and Joint Embedding Model) implementation. This is done in order to be able to apply transfer Learning by importing the model parameters from a previous **Proposed Model** training run on the Sorel-20m dataset. These parameters can be then '*fine-tuned*' (further optimized) on the new family classification task using samples from the fresh dataset. On the other hand, the *Family Classifier* output head, which is built on top of the *PE Embedding* layer, is new and its parameters are therefore always learned from scratch.

The *Family Classifier* topology, on the other hand, consists of 5 dense blocks of output sizes 128, 256, 128, 64 and 10, respectively. These dense blocks are designed in the same way as the ones used in the *Proposed Model* base topology - a linear layer with dropout, followed by batch normalization and ReLU activation function - but for the last one which simply consists of a simple linear layer trailed by the *Sigmoid* non-linearity. As always, at network definition it is possible to dynamically set the hyper-parameters for this topology (as well as the shared base topology) such as the exact number of layers, the linear layer's sizes, the dropout probability p_d, the normalization function and the activation function used. However, most of the hyper-parameters for the *Family classifier* topology are shared with the model *shared base* with the exception of the number of layers and the layers' dimensions, which can be independently set. The *Family classifier* output head transforms a given input sample PE embedding (of size 32) into a set of 10 class scores (one per family) which can be transformed into family label probabilities through the use of the *softmax* function. The final family prediction, for each sample, is then computed by taking the family label with the highest score. The loss between the family class output scores $f_{fam}(x)$ and the ground truth family label y_{fam}, on the other hand, is computed

using the Pytorch *Cross Entropy Loss* function. This corresponds to computing the mean of the per-sample losses as depicted by equation 6.1.

$$L = \frac{\sum_{i=1}^{N} L(i, y_{fam_i})}{N} \tag{6.1}$$

where N is the total number of samples for the current training mini-batch and $\mathcal{L}(i, y_{fam_i})$ is the loss for sample i having ground truth family label y_{fam_i} as defined in equation 6.2.

$$L(x, fam) = -\log\left(\frac{e^{x[fam]}}{\sum_j e^{x[j]}}\right) = -x[fam] + \log\left(\sum_j e^{x[j]}\right) \tag{6.2}$$

where $x[j]$ denotes the sample class score for family j.

Net Definition

Algorithm 46 Malware Family Classifier Net class, Init part 1

```
1:  class NET
2:      function INIT(self, families, feature_dimension, embedding_dimension, layer_sizes,
        fam_class_layer_sizes, dropout_p, activation_function, normalization_function)
3:
4:          self.families ← families
5:          self.n_families ← len(families)
6:          self.encoded_families ← [idx ∀ idx ∈ range(self.n_families)]
7:          self.embedding_dimension ← embedding_dimension
8:          self.loss_criterion ← nn.CrossEntropyLoss()
9:          layers ← [ ]
10:         fam_class_layers ← [ ]
11:         if layer_sizes is None then
12:             layer_sizes ← [512, 512, 128]
13:         end if
14:
15:         if fam_class_layer_sizes is None then
16:             fam_class_layer_sizes ← [64, 32]
17:         end if
18:
19:         for all i, ls ∈ enumerate(layer_sizes) do
20:             if i = 0 then
21:                 layers.append(nn.Linear(feature_dimension, ls))
22:             else
23:                 layers.append(nn.Linear(layer_sizes[i −1], ls))
24:             end if
25:
26:             layers.append(normalization_function(ls))
27:             layers.append(activation_function())
28:             layers.append(nn.Dropout(dropout_p))
29:         end for
30:
31:         self.model_base ← nn.Sequential(*tuple(layers))
32:         self.sigmoid ← nn.Sigmoid()
33:                                                    ▷ Continues on next page..
```

Algorithm 46-47 (Malware Family Classifier Init function) shows how the Family Classifier model is initialized. In particular the function first defines the model *shared base* topology (*model_base*) as a sequence of blocks composed each of a linear layer, a Normalization layer,

Algorithm 47 Malware Family Classifier Net class, Init part 2

```
34:                                                    ▷ Continued from previous page..
35:         self.pe_embedding ← nn.Sequential(
36:             nn.Linear(layer_sizes[-1], self.embedding_dimension),
37:             normalization_function(self.embedding_dimension),
38:             activation_function())
39:
40:         for all i, ls ∈ enumerate(fam_class_layer_sizes) do
41:             if i = 0 then
42:                 fam_class_layers.append(nn.Linear(embedding_dimension, ls))
43:             else
44:                 fam_class_layers.append(nn.Linear(layer_sizes[i −1], ls))
45:             end if
46:
47:             fam_class_layers.append(normalization_function(ls))
48:             layers.append(activation_function())
49:             layers.append(nn.Dropout(dropout_p))
50:         end for
51:
52:         fam_class_layers.append(nn.Linear(fam_class_layer_sizes[-1], self.n_families))
53:         self.family_classifier ← nn.Sequential(*tuple(fam_class_layers))
54:         self.softmax_output ← nn.Softmax(dim=1)
55:     end function
```

Algorithm 48 Malware Family Classifier Net class, Forward

```
56:     function FORWARD(self, data)
57:         base_out ← self.model_base(data)
58:         pe_embedding ← self.pe_embedding(base_out)
59:         scores ← self.families_classifier(pe_embedding)
60:         rv ← {'scores': scores,
61:               'probs': self.softmax_output(scores)}
62:
63:         return rv
64:     end function
```

Algorithm 49 Malware Family Classifier Net class, Compute Loss

```
65:     function COMPUTELOSS(self, predictions, labels, loss_wts)
66:
67:         return self.loss_criterion(predictions['scores'], labels)
68:     end function
69: end class
```

an Activation function and dropout, exactly as previously done for the ***Proposed Model*** implementation. Next, the *pe_embedding* topology is constructed as a single linear layer with output size equal to the *embedding_dimension* (= 32), followed by a Normalization layer and an Activation function. Thereafter, the function defines the *Family Classifier* topology as a sequence of dense blocks - equal to the ones used in the *shared base* part - trailed by a final linear layer of output size equal to the number of family labels (= 10) and, optionally, a *Softmax* non-linearity.

Algorithm 48 (***Malware Family Classifier* forward function**), on the other hand, shows how the input data (PE files' numerical feature vectors) is forwarded through the model. In particular, the function feeds the model base topology (*model_base*) with the input data producing an intermediate representation *base_out*. This intermediate representation is then forwarded to the *pe_embedding* subnet which outputs the input samples' PE latent representation (of size 32). At the beginning of the family classifier model training procedure, the mapping function between the samples' input features and the produced PE embeddings corresponds, if transfer Learning was

applied, to the one learned by the **Proposed Model** using the Sorel-20m dataset. The output class (family) scores are computed by feeding the *Family classifier* topology with the produced input samples' PE embeddings. Finally, the family probabilities are simply obtained by applying the *Softmax* function to the previously computed family *scores*.

Finally, as presented by algorithm 49 (***Malware Family Classifier* compute loss function**) the loss between the predicted family scores and the ground truth family label is simply computed by applying the selected loss criterion, which corresponds in this case to the Pytorch **cross_entropy_loss** function.

6.2.2 Experiments

The resulting model was trained (and validated) for 25 *epochs* on the training and validation sub-splits of the Fresh Dataset using different numbers of layers and numbers of nodes per layer (layer sizes) for the Family classifier additional output head while keeping the same model shared base topology inherited by the previously described **Proposed Model** implementation. Moreover, various combinations of learning rates (LR), momentum, weight decay and loss weights were used in separate family classifier training runs. The model which performed best during validation was then evaluated on the test sub-split of the Fresh Dataset producing the corresponding accuracy and multiclass classification '*micro*' (global average), '*macro*' (unweighted per-class average) and '*weighted*' (per-class average, weighted by support) scores such as *Jaccard Similarity*, *recall*, *precision*, *f1* score, as well as the resulting '*OVO*' (One Vs. One) and '*OVR*' (One Vs. Rest) *AUC-ROC* scores.

The final Family Classifier head architecture consisted of 5 layers of sizes 128, 256, 128, 64, and 10 respectively, with dropout probability $p_d = 5\%$ and **ReLU** activation function and was trained with a learning rate $LR = 10^{-3}$, **Adam** optimizer, $momentum = 10^{-3}$, $weight\ decay = 0.01$ and $batch\ size = 250$.

The evaluation results of the *Family Classifier* model implementation in the Malware Family classification task are reported in chapter 7.

6.2.3 Family Classifier Training and Evaluation algorithms

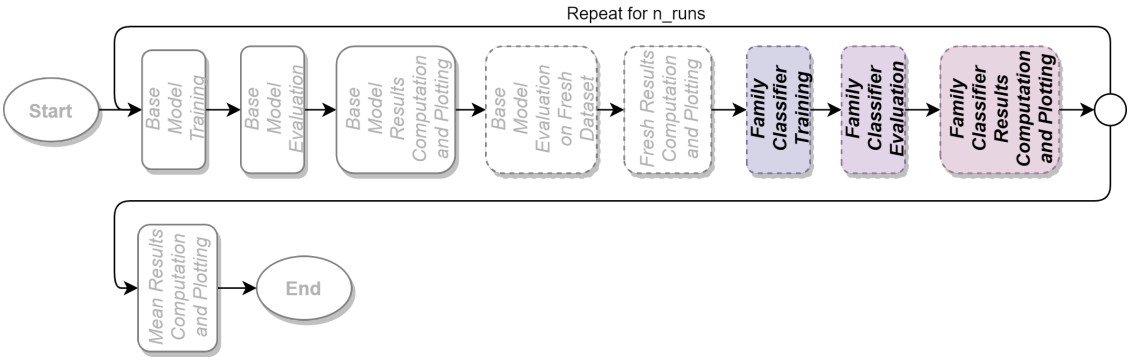


Figure 6.7: Family Classifier Training and Evaluation Workflow steps

Algorithms 50-51 (**Family Classifier Train function**) and 52 (**Family Classifier Evaluate function**) show how the ***Family Classifier*** model implementation gets trained and evaluated using samples from the Fresh Dataset.

Family Classifier Training

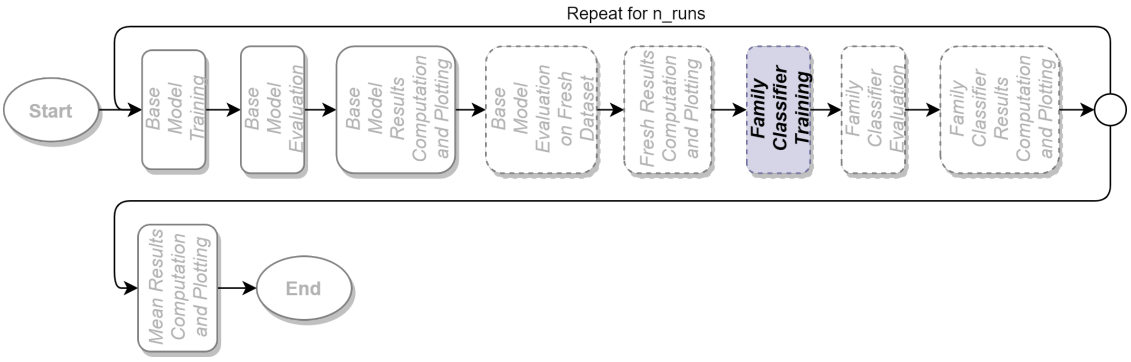


Figure 6.8: Family Classifier Training Workflow step

Algorithm 50 Train Family Classifier function, part 1

```
1:  function TrainNetwork(fresh_ds_path, checkpoint_path, batch_size, epochs,
    train_split_proportion, valid_split_proportion, test_split_proportion, workers, ...)
2:
3:      split_proportions ← [train_split_proportion,
4:                           valid_split_proportion,
5:                           test_split_proportion]
6:
7:      train_gen, valid_gen, _ ← get_generator(fresh_ds_path,
8:          splits=split_proportions,
9:          batch_size=batch_size,
10:         return_shas=True,
11:         num_workers=workers, ..)
12:
13:     model ← Family_Net(families=[label_to_sig(lab) ∀ lab ∈ range(n_families)],
14:         feature_dimension=2381,
15:         embedding_dimension=32,
16:         layer_sizes=run_params['layer_sizes'],
17:         fam_class_layer_sizes=run_params['fam_class_layer_sizes'],
18:         dropout_p=run_params['dropout_p'],
19:         activation_function=run_params['activation_function'],
20:         normalization_function=run_params['normalization_function'])
21:
22:     if checkpoint_path is not None then
23:         model.load_state_dict(checkpoint_path, strict=False)
24:         parameters_to_optimize ← [
25:             {'params': model.family_classifier.parameters()},
26:             {'params': model.pe_embedding.parameters(), 'lr': run_params['lr'] / 10},
27:             {'params': model.model_base.parameters(), 'lr': run_params['lr'] / 10}
28:         ]
29:     else
30:         parameters_to_optimize ← model.parameters()
31:     end if
32:
33:     opt ← run_params['optimizer'](parameters_to_optimize, run_params['lr'],
34:                                   run_params['weight_decay'], run_params['momentum'])
35:     scheduler ← MultiStepLR(opt, milestones=[(3 × epochs) // 4], gamma=0.1)
36:                                                          ▷ Continues on next page..
```

The *Family Classifier* training function, as shown in alg. 50-51, first opens and loads the Fresh

Algorithm 51 Train Family Classifier function, part 2

```
                                                    ▷ Continued from previous page..
    for all epoch ∈ range(epochs) do
        model.train()

        for all shas, features, labels ∈ train_gen do
            opt.zero_grad
            out ← model(features)
            loss ← model.ComputeLoss(out, labels)
            preds ← [argmax(line ∀ line ∈ out['scores'])]
            accuracy ← sum(1 ∀ i, pred ∈ enumerate(preds) if pred = labels[i]) / len(labels)
            log(loss)
            log(accuracy)
            loss.backwawrd()
            opt.step()
        end for

        scheduler.step()
        model.eval()

        for all shas, features, labels ∈ valid_gen do
            torch.no_grad()
            out ← model(features)
            loss ← model.ComputeLoss(out, labels)
            preds ← [argmax(line ∀ line ∈ out['scores'])]
            accuracy ← sum(1 ∀ i, pred ∈ enumerate(preds) if pred = labels[i]) / len(labels)
            log(loss)
            log(accuracy)
        end for
    end for
end function
```

Dataset splitting it in training, validation and test subsets following the selected proportions. The *Family Classifier* model then gets instantiated by providing the necessary parameters to its *init* function. Next, if a checkpoint of a **Proposed Model** training run is provided, the state parameters for the *shared base* topology and *PE embedding* layer are loaded into the newly instantiated **Family Classifier** model and the learning rates for the different model subnets are set such that the *shared base* and *PE embedding* topologies are just '*fine-tuned*' while training the model (by using a smaller learning rate). Then, the selected optimizer (**Adam** or **SGD**) gets instantiated given the model parameters, selected learning rate, weight decay and momentum. At this point a scheduler is also instantiated with the purpose of modifying the model's parameters learning rates (multiplying them by 0.1) after 3/4 of the total number of epochs. After these preliminary steps, the function trains (and validates) the **Family Classifier** model for '*epochs*' epochs. More specifically, at each epoch, the model is set to training mode and then, for all the mini-batches of data randomly sampled from the (fresh dataset) training generator, the input features are fed into the model to obtain the corresponding output class (family) scores. Next, the loss between the predicted class scores and the ground truth family label is computed and used to produce the gradients which are back-propagated through the network. Moreover, the function also computes the predicted family label - by selecting, for each sample, the family label with maximum class score - and the model accuracy for the current mini-batch - by dividing the number of correct predictions by the total amount of predictions. Then, the previously computed model loss and accuracy are logged and the model is set to *validation* mode after having updated the scheduler. Finally, for each mini-batch of data from the (fresh dataset) validation generator, the function computes and logs the model loss and accuracy.

Family Classifier Evaluation

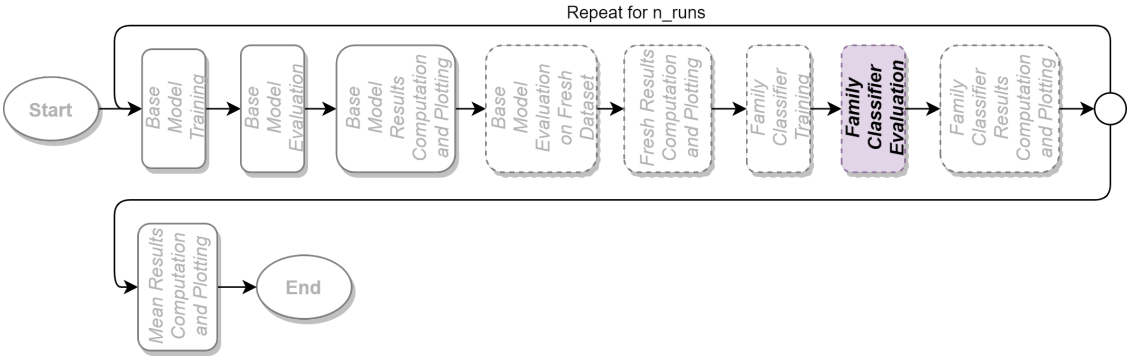


Figure 6.9: Family Classifier Evaluate Workflow step

Algorithm 52 Evaluate Family Classifier function

```
1: function EvaluateNetwork(fresh_ds_path, checkpoint_path, batch_size,
   train_split_proportion, valid_split_proportion, test_split_proportion, workers, ...)
2:
3:     split_proportions ← [train_split_proportion,
4:                          valid_split_proportion,
5:                          test_split_proportion]
6:
7:     _, _, test_gen ← get_generator(fresh_ds_path,
8:         splits=split_proportions,
9:         batch_size=batch_size,
10:        return_shas=True,
11:        num_workers=workers, ..)
12:
13:    model ← Family_Net(families=[label_to_sig(lab) ∀ lab ∈ range(n_families)],
14:        feature_dimension=2381,
15:        embedding_dimension=32,
16:        layer_sizes=run_params['layer_sizes'],
17:        fam_class_layer_sizes=run_params['fam_class_layer_sizes'],
18:        dropout_p=run_params['dropout_p'],
19:        activation_function=run_params['activation_function'],
20:        normalization_function=run_params['normalization_function'])
21:
22:    model.load_state_dict(checkpoint_path)
23:    model.eval()
24:
25:    for all shas, features, labels ∈ test_gen do
26:        out ← model(features)
27:        preds ← [max(line ∀ line ∈ out['scores'])]
28:        accuracy ← sum(1 ∀ i, pred ∈ enumerate(preds) if pred = labels[i]) / len(labels)
29:        log(accuracy)
30:        save_to_file(labels, out['probs'])
31:    end for
32: end function
```

Algorithm 52 (**Family Classifier Evaluation function**) shows how the **Family Classifier** model evaluation function is implemented in Python/Pytorch code. In particular, the function first opens and loads the Fresh Dataset splitting it in training, validation and test subsets following the selected proportions, as done by the training function. The *Family Classifier* model then gets instantiated by providing the necessary arguments to its *init* function and the checkpoint state

parameters from a previous training run are loaded. Thereafter, the model is set into *evaluation* mode and, for all the mini-batches of data from the (Fresh Dataset) test generator, the model predictions, overall accuracy and family label probabilities are computed and saved to file/logged.

Family Classifier Results Computation and Plotting

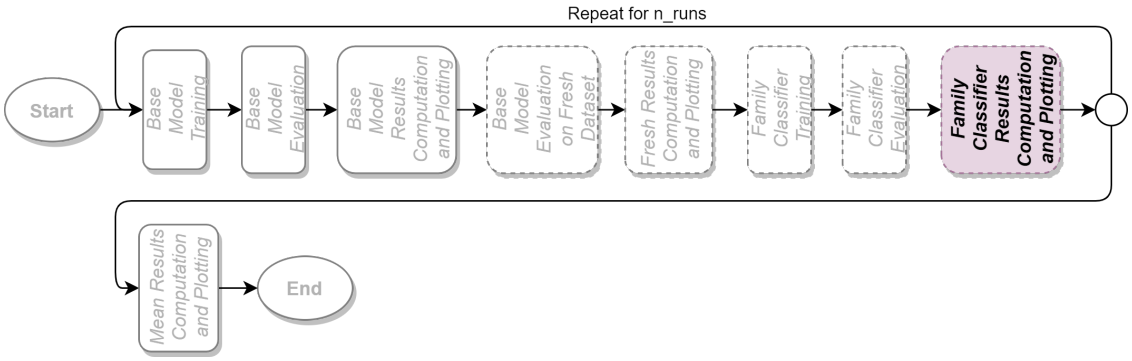


Figure 6.10: Family Classifier Results Computation and Plotting Workflow step

The Family classifier evaluation ***micro*** (global average), ***macro*** (unweighted per-family average) and ***weighted*** (per-family average, weighted by support) scores, such as '*Jaccard similarity score*', '*recall*', '*precision*' and '*f1-score*', are computed and saved to file, together with the resulting **OVO** (One Vs. One) and **OVR** (One Vs. Rest) AUC-ROC scores and confusion matrix, by a distinct set of functions - which simply apply the respective mathematical formulas - not presented in this document.

Chapter 7

Results

This chapter presents the evaluation results of the different models implemented. In particular, in the first section (7.1) are depicted the different models' results on the task of malware detection (Malicious/Benign label prediction). The same models' evaluation results on the task of malware description via SMART tags are presented in a separate section (7.2). The next section (7.3), presents the models' (and original features') results on the family prediction and ranking tasks using samples from the Fresh Dataset. The final section (7.4), on the other hand, presents the evaluation results of the Malware Family Classifier on the family classification task.

7.1 Malware Detection results

In this section the models' results concerning the **Malicious/Benign** label prediction task are presented. In particular for each model are shown the corresponding **AUC-ROC** (Area Under ROC Curve) score (table 7.1) and the various binary cross entropy loss statistics (**TPR** (True Positive Rate), **Accuracy**, **Recall**, **Precision** and **F1 Score**) at different **FPRs** (False Positive Rates) (table 7.2). Moreover, for the **TPR** statistic, also the **Proposed Model** Error and Standard Deviation Reductions with respect to the other previous models are presented, in percentage. Next, the **ROC** (Receiving Operating Characteristic) curves of the different models are presented in figures 7.1, 7.2 and 7.3.

The models considered in this section are:

- **ALOHA (M/B only)**: *ALOHA* model implementation described in section 5.1.1, with only the *Malicious/Benign* label prediction head enabled. This model thus represents a simple single task FNN designed exclusively for the malicious label prediction task.
- **ALOHA**: final *ALOHA* model implementation described in section 5.1.1, with all the additional output heads enabled.
- **Proposed Model**: final *Proposed Model* implementation described in section 6.1.1.

Malware Label	ALOHA (M/B only)	ALOHA	Proposed Model
AUC-ROC	0.995±0.000	0.995±0.001	**0.996±0.001**

Table 7.1: AUC-ROC (Area Under Curve) of the different models for the **Malware Label** prediction task. Results were aggregated over **2** training runs with different weight initializations and minibatch orderings. Best results are shown in **bold**.

Malware Label	FPR				
	10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
TPR					
ALOHA (M/B only)	0.215±0.032	0.647±0.033	0.883±0.005	0.959±0.003	0.990±0.001
ALOHA	0.599±0.023	0.790±0.005	0.879±0.007	0.956±0.011	0.990±0.001
Proposed Model	**0.622±0.058**	**0.836±0.007**	**0.909±0.001**	**0.959±0.001**	**0.992±0.001**
Error Reduction wrt ALOHA (M/B only)	51.8%	53.5%	22.2%	0.0%	20.0%
Error Reduction wrt ALOHA	5.7%	21.9%	24.8%	6.8%	20.0%
Std Reduction wrt ALOHA (M/B only)	-81.2%	78.8%	80.0%	66.7%	0.0%
Std Reduction wrt ALOHA	-152.2%	-40.0%	85.7%	90.9%	0.0%
Accuracy					
ALOHA (M/B only)	0.696±0.012	0.863±0.013	0.954±0.002	0.978±0.001	**0.935±0.000**
ALOHA	0.845±0.009	0.919±0.002	0.953±0.003	0.977±0.004	**0.935±0.000**
Proposed Model	**0.854±0.023**	**0.936±0.003**	**0.964±0.000**	**0.978±0.000**	**0.935±0.000**
Recall					
ALOHA (M/B only)	0.215±0.032	0.647±0.033	0.883±0.005	0.959±0.003	0.990±0.001
ALOHA	0.599±0.023	0.790±0.005	0.879±0.007	0.956±0.011	0.990±0.001
Proposed Model	**0.622±0.058**	**0.836±0.007**	**0.909±0.001**	**0.959±0.001**	**0.992±0.001**
Precision					
ALOHA (M/B only)	**1.000±0.000**	**1.000±0.000**	**0.998±0.000**	**0.984±0.000**	**0.862±0.000**
ALOHA	**1.000±0.000**	**1.000±0.000**	**0.998±0.000**	**0.984±0.000**	**0.862±0.000**
Proposed Model	**1.000±0.000**	**1.000±0.000**	**0.998±0.000**	**0.984±0.000**	**0.862±0.000**
F1 Score					
ALOHA (M/B only)	0.353±0.043	0.785±0.024	0.937±0.003	0.971±0.001	0.921±0.000
ALOHA	0.749±0.018	0.883±0.003	0.935±0.004	0.970±0.006	**0.922±0.000**
Proposed Model	**0.765±0.044**	**0.911±0.004**	**0.952±0.001**	**0.971±0.000**	0.922±0.001

Table 7.2: Mean and standard deviation results (TPR, Accuracy, Recall, Precision and F1-Score) of the different models for the **Malware Label** prediction task at different **FPR**s (*False Positive Rates*). Results were aggregated over **2** training runs with different weight initializations and minibatch orderings. Best results are shown in **bold**. Under **TPR** results are also presented the percentage reduction in mean detection error and in ROC curve standard deviation introduced by the *Proposed Model* with respect to both *ALOHA* model and *Joint Embedding*.

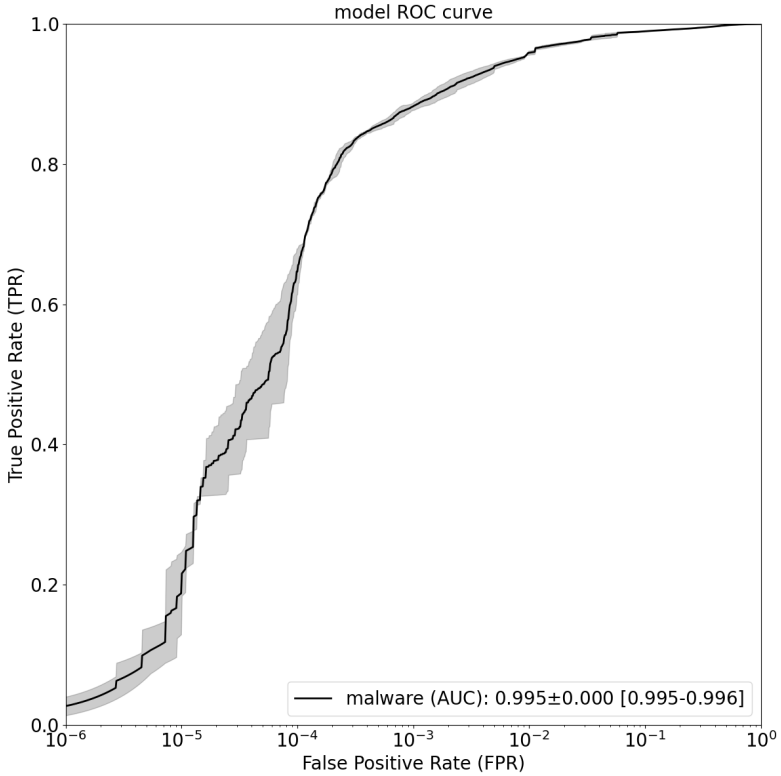


Figure 7.1: ROC curve and AUC statistics of **ALOHA (M/B only)** model for the **Malware Label**. The line represents the *mean* TPR at a given FPR, while the shaded region represents the *standard deviation*. Statistics were computed over **2** training runs, each with random parameter initialization.

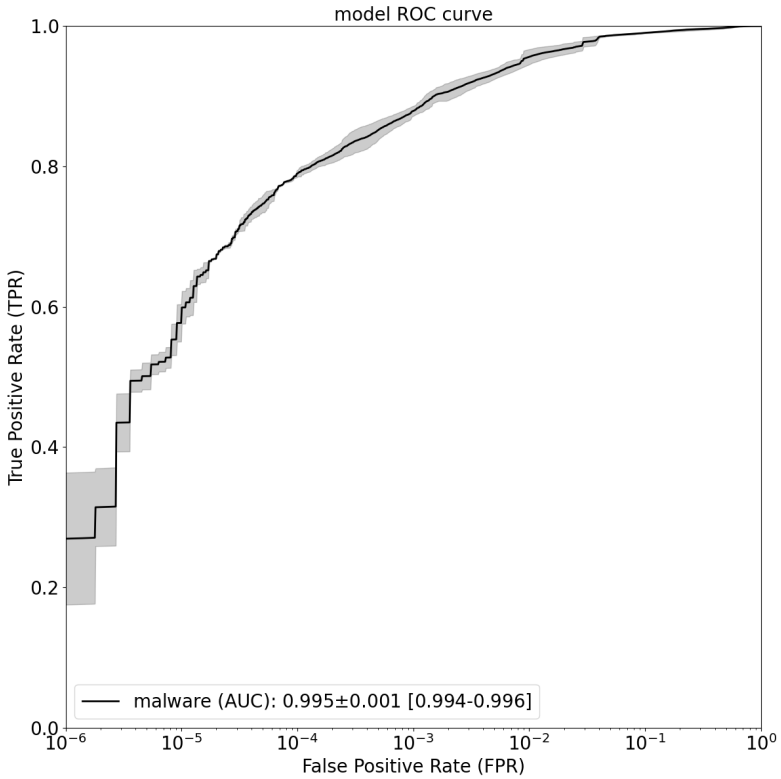


Figure 7.2: ROC curve and AUC statistics of **ALOHA** model for the **Malware Label**. The line represents the *mean* TPR at a given FPR, while the shaded region represents the *standard deviation*. Statistics were computed over **2** training runs, each with random parameter initialization.

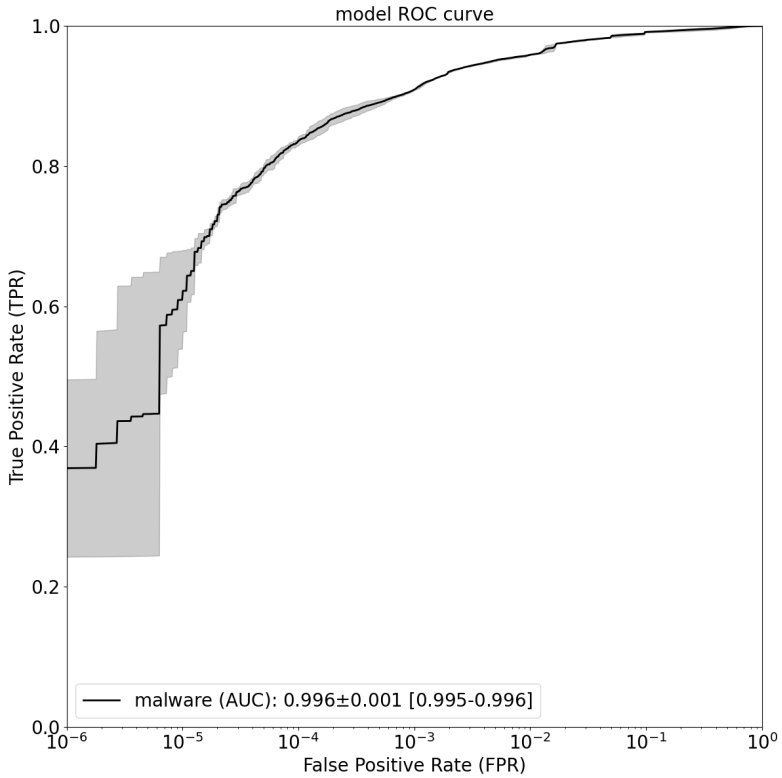


Figure 7.3: ROC curve and AUC statistics of **Proposed Model** for the **Malware Label**. The line represents the *mean* TPR at a given FPR, while the shaded region represents the *standard deviation*. Statistics were computed over **2** training runs, each with random parameter initialization.

7.1.1 Summary

Here (table 7.3) it is presented a brief summary of the previously mentioned statistics for the different models at **FPR** = 1%.

Malware Label (at FPR = 1%)					
Model	TPR	Accuracy	Precision	Recall	F1 score
ALOHA (M/B only)	0.959±0.003	0.978±0.001	**0.984±0.000**	0.959±0.003	0.971±0.001
ALOHA	0.956±0.011	0.977±0.004	**0.984±0.000**	0.956±0.011	0.970±0.006
Proposed Model	**0.959±0.001**	**0.978±0.000**	**0.984±0.000**	**0.959±0.001**	**0.971±0.000**

Table 7.3: Summary of the mean and standard deviation results of the different models for the **Malware Label** prediction task at **FPR** = 1%. Results were aggregated over **2** training runs with different weight initializations and minibatch orderings. Best results are shown in **bold**.

7.1.2 Comments

As shown in table 7.1, the **Proposed Model** implementation provided a slightly better *AUC-ROC* score than both the **ALOHA** model implementations when tested on the malware detection - Malicious/Benign label prediction - task. Moreover, table 7.2 shows that the **Proposed Model** implementation consistently provided better scores for all the binary cross entropy statistics used as performance measures than the other two models at different *FPRs* (False Positive Rates). Most importantly, the **Proposed Model** implementation performed better than the other models at $FPR = 1\%$, as summarized in table 7.3. These results prove the effectiveness of using the 'harder' problem of creating a *Joint Embedding* space where samples are close to their associated SMART tags as an additional task to improve the performance of the model at the malware detection task.

7.2 Malware Description via SMART tags results

In this section the models' evaluation results for the SMART tagging prediction task are presented. More specifically, table 7.4 shows, for each predicted tag, the **AUC-ROC** scores relative to the different models. Table 7.5, on the other hand, presents the per-tag binary cross entropy statistics at **FPR** = 1%. Finally, figures 7.4, 7.5 and 7.6 present, for each model, a single graph showing the mean **ROC** curves of all predicted tags plus the Malware label (if available).

The models considered in this section are:

- **ALOHA**: final *ALOHA* model implementation described in section 5.1.1, with all the additional output heads enabled.
- **Joint Embedding**: final *Joint Embedding* model implementation described in section 5.1.2.
- **Proposed Model**: final *Proposed Model* implementation described in section 6.1.1.

AUC-ROC	ALOHA	Joint Embedding	Proposed Model
Adware Tag	0.969±0.004	0.975±0.001	**0.976±0.000**
Crypto-miner Tag	0.989±0.001	**0.993±0.001**	0.989±0.000
Downloader Tag	0.967±0.002	0.980±0.002	**0.983±0.002**
Dropper Tag	0.973±0.001	0.976±0.001	**0.979±0.000**
File-infector Tag	0.985±0.000	0.982±0.003	**0.987±0.000**
Flooder Tag	0.985±0.001	**0.985±0.000**	**0.985±0.000**
Installer Tag	0.971±0.004	0.973±0.002	**0.981±0.000**
Packed Tag	0.980±0.003	0.981±0.001	**0.983±0.001**
Ransomware Tag	0.980±0.008	**0.990±0.002**	0.985±0.003
Spyware Tag	0.961±0.002	**0.975±0.004**	0.973±0.003
Worm Tag	**0.975±0.007**	0.957±0.012	0.964±0.002

Table 7.4: Mean and standard deviation AUC-ROC (Area Under Curve) of the different models for the the prediction of the different Tags. Results were aggregated over **3** training runs with different weight initializations and minibatch orderings. Best results are shown in **bold**.

	TPR	Accuracy	Precision	Recall	F1 score
Adware Tag (at FPR = 1%)					
ALOHA	0.682±0.039	0.972±0.002	0.806±0.009	0.682±0.039	0.738±0.026
Joint Embedding	0.688±0.004	**0.973±0.000**	0.808±0.001	0.688±0.004	0.743±0.002
Proposed Model	**0.701±0.015**	0.973±0.001	**0.811±0.003**	**0.701±0.015**	**0.752±0.010**
Crypto-miner Tag (at FPR = 1%)					
ALOHA	0.761±0.138	0.987±0.002	0.512±0.046	0.761±0.138	0.611±0.077
Joint Embedding	**0.926±0.001**	**0.989±0.000**	**0.565±0.000**	**0.926±0.001**	**0.701±0.000**
Proposed Model	0.746±0.159	0.987±0.002	0.505±0.054	0.746±0.159	0.601±0.090
Downloader Tag (at FPR = 1%)					
ALOHA	0.599±0.018	0.957±0.002	0.845±0.004	0.599±0.018	0.701±0.014

...continued on next page

...continued from previous page

	TPR	Accuracy	Precision	Recall	F1 score
Joint Embedding	0.665±0.006	0.963±0.000	0.859±0.001	0.665±0.006	0.750±0.004
Proposed Model	**0.691±0.008**	**0.965±0.001**	**0.863±0.001**	**0.691±0.008**	**0.767±0.005**
Dropper Tag (at FPR = 1%)					
ALOHA	0.663±0.045	0.948±0.006	0.906±0.006	0.663±0.045	0.765±0.032
Joint Embedding	**0.735±0.022**	0.957±0.003	**0.915±0.002**	**0.735±0.022**	**0.815±0.014**
Proposed Model	0.729±0.013	**0.957±0.002**	0.914±0.001	0.729±0.013	0.811±0.008
File-infector Tag (at FPR = 1%)					
ALOHA	0.844±0.001	0.966±0.000	0.942±0.000	0.844±0.001	0.890±0.000
Joint Embedding	**0.863±0.002**	**0.970±0.000**	**0.943±0.000**	**0.863±0.002**	**0.901±0.001**
Proposed Model	0.854±0.003	0.968±0.000	**0.943±0.000**	0.854±0.003	0.896±0.002
Flooder Tag (at FPR = 1%)					
ALOHA	0.904±0.002	**0.990±0.000**	**0.141±0.000**	0.904±0.002	0.243±0.000
Joint Embedding	0.905±0.005	**0.990±0.000**	0.141±0.001	0.905±0.005	**0.244±0.001**
Proposed Model	**0.905±0.002**	**0.990±0.000**	**0.141±0.000**	**0.905±0.002**	**0.244±0.001**
Installer Tag (at FPR = 1%)					
ALOHA	0.724±0.044	0.985±0.001	0.567±0.015	0.724±0.044	0.636±0.026
Joint Embedding	0.736±0.001	0.985±0.000	0.571±0.000	0.736±0.001	0.643±0.000
Proposed Model	**0.773±0.009**	**0.986±0.000**	**0.583±0.003**	**0.773±0.009**	**0.665±0.005**
Packed Tag (at FPR = 1%)					
ALOHA	0.710±0.015	0.952±0.002	0.918±0.002	0.710±0.015	0.801±0.010
Joint Embedding	0.749±0.032	0.957±0.004	0.922±0.003	0.749±0.032	0.826±0.021
Proposed Model	**0.766±0.010**	**0.959±0.001**	**0.924±0.001**	**0.766±0.010**	**0.838±0.007**
Ransomware Tag (at FPR = 1%)					
ALOHA	0.855±0.001	**0.983±0.000**	0.825±0.000	0.855±0.001	0.840±0.001
Joint Embedding	**0.861±0.003**	**0.983±0.000**	**0.826±0.001**	**0.861±0.003**	**0.843±0.002**
Proposed Model	0.855±0.000	**0.983±0.000**	0.825±0.000	0.855±0.000	0.840±0.000
Spyware Tag (at FPR = 1%)					
ALOHA	0.599±0.027	0.948±0.003	0.877±0.005	0.599±0.027	0.712±0.021
Joint Embedding	**0.724±0.004**	**0.962±0.000**	**0.897±0.000**	**0.724±0.004**	**0.801±0.002**
Proposed Model	0.713±0.004	0.960±0.000	0.895±0.000	0.713±0.004	0.793±0.003
Worm Tag (at FPR = 1%)					
ALOHA	0.611±0.017	0.930±0.003	0.920±0.002	0.611±0.017	0.734±0.013
Joint Embedding	0.663±0.008	0.938±0.001	**0.926±0.001**	0.663±0.008	0.773±0.006
Proposed Model	**0.669±0.008**	**0.939±0.001**	**0.926±0.001**	**0.669±0.008**	**0.777±0.006**

Table 7.5: Summary of the mean and standard deviation results of the different models for the prediction of the different tags at **FPR** = 1%. Results were aggregated over **3** training runs with different weight initializations and minibatch orderings. Best results are shown in **bold**.

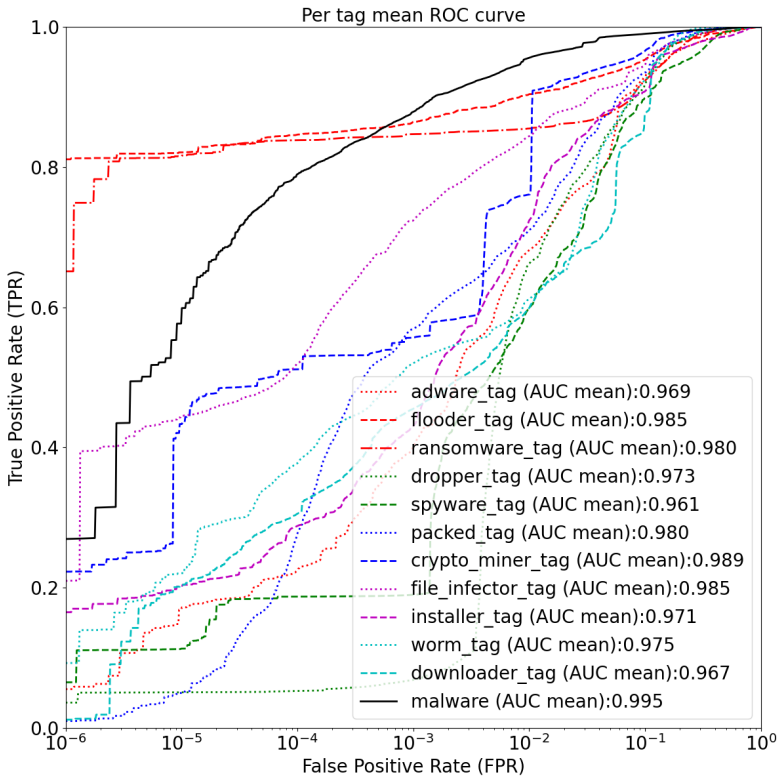


Figure 7.4: Mean ROC curve and AUC statistics of **ALOHA** model for the tags/labels. The line represents the *mean* TPR at a given FPR. Statistics were computed over **2** training runs, each with random parameter initialization.

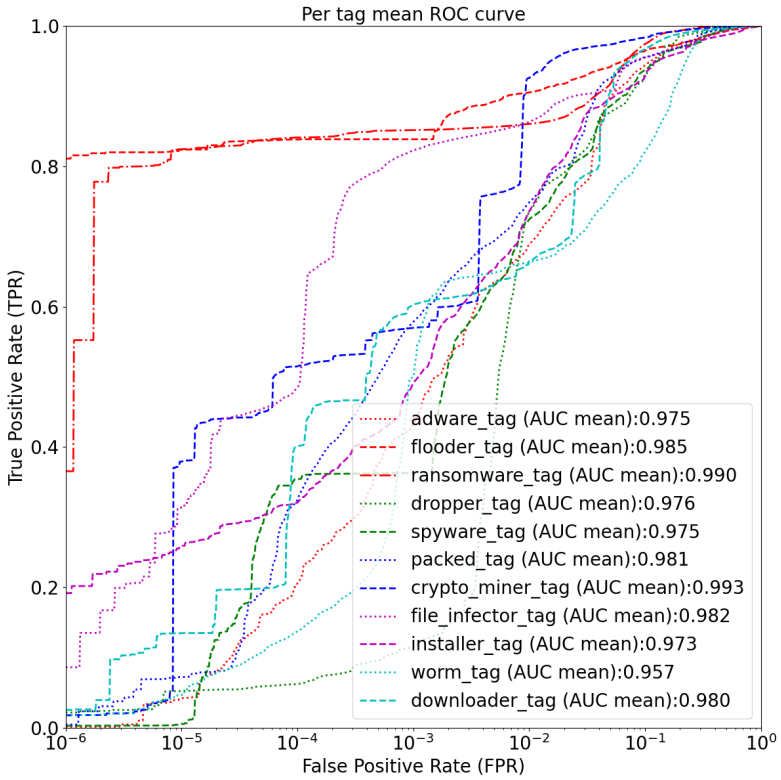


Figure 7.5: Mean ROC curve and AUC statistics of **Joint Embedding** model for all tags/labels. The line represents the *mean* TPR at a given FPR. Statistics were computed over **2** training runs, each with random parameter initialization.

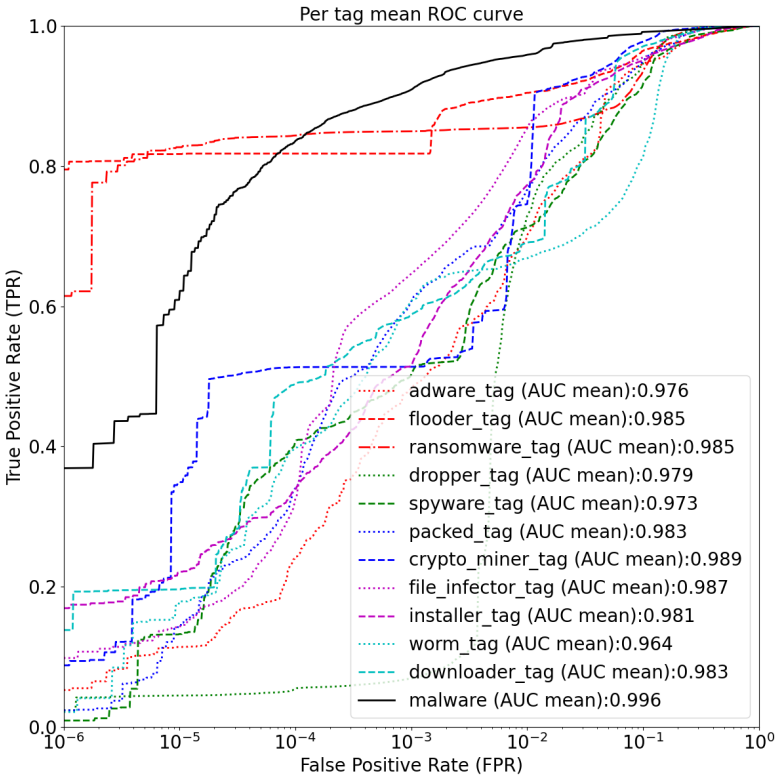


Figure 7.6: Mean ROC curve and AUC statistics of **Proposed Model** for the tags/labels. The line represents the *mean* TPR at a given FPR. Statistics were computed over **2** training runs, each with random parameter initialization.

7.2.1 Mean per-sample tagging scores

Here (table 7.6) are presented the models' *Mean per-sample scores*, more specifically the **Jaccard Similarity** and **Mean per-Sample Accuracy** scores, achieved by the models at different **FPRs**.

Mean per-sample tagging scores	**FPR** 10^{-5}	10^{-4}	10^{-3}	10^{-2}	10^{-1}
Jaccard Similarity					
ALOHA	**0.711±0.012**	0.748±0.010	0.799±0.006	0.864±0.001	0.794±0.014
Joint Embedding	0.684±0.009	0.742±0.009	0.820±0.005	0.884±0.001	0.785±0.007
Proposed Model	0.695±0.019	**0.761±0.008**	**0.826±0.003**	**0.885±0.001**	**0.799±0.002**
Mean per-Sample Accuracy					
ALOHA	0.646±0.006	0.690±0.001	0.749±0.000	0.796±0.001	**0.699±0.020**
Joint Embedding	0.640±0.001	0.684±0.005	0.761±0.005	0.810±0.000	0.684±0.008
Proposed Model	**0.649±0.010**	**0.703±0.010**	**0.773±0.003**	**0.813±0.000**	0.695±0.001

Table 7.6: Mean and standard deviation of mean per-sample tagging results (*Jaccard simialrity* and *mean per-sample accuracy*) for the different models. Results were aggregated over **2** training runs with different weight initializations and minibatch orderings. Best results are shown in **bold**.

7.2.2 Comments

As shown in table 7.4, the **Proposed Model** implementation provided better *AUC-ROC* scores than the **ALOHA** and **Joint Embedding** model implementations for most of the target SMART

tags. More specifically, it performed better when dealing with the *Adware*, *Downloader*, *Dropper*, *File-infector*, *Flooder*, *Installer* and *Packed* tags. The **Joint Embedding** model implementation, on the other hand provided a higher *AUC-ROC* score than the other models for the *Crypto-miner*, *Flooder* (on par with the **Proposed Model** implementation), *Ransomware* and *Spyware* tags. Finally, the **ALOHA** model implementation performed better than the other models only on the *Worm* tag. It is interesting to notice, however, that the **Proposed Model** implementation results for the tags on which it did not perform best, were always the second best and anyway comparable to the best results.

On top of that, table 7.5 shows that the **Proposed Model** implementation provided the best scores for all the binary cross entropy statistics used as performance measures (at $FPR = 1\%$) when dealing with the *Downloader*, *Flooder*, *Installer*, *Packed* and *Worm* tags. It also provided the best *TPR*, *Precision*, *Recall* and *F1 score* at $FPR = 1\%$ for the *Adware* tag, while having the same *accuracy* than the **Joint Embedding** model implementation but with a higher standard deviation. On the other hand, the **Proposed Model** implementation provided worse scores for all binary cross entropy statistics than the **Joint Embedding** model implementation for the *Crypto-miner*, *File-infector* (for which it provided the same *precision*), *Ransomware* (for which it provided the same *accuracy*) and *Spyware* tags. Finally, the **Proposed Model** implementation provided in general worse results than the **Joint Embedding** model implementation for the *Dropper* tag but for the *Accuracy* score, for which it provided the same mean value with a lower standard deviation. Again, it is interesting to notice that when the **Proposed Model** did not provide the best results it was always the second best (and the scores were comparable).

Finally, table 7.6 shows that the **Proposed Model** implementation provided, in general, better *Jaccard Similarity* and *Mean per-Sample Accuracy* than the **ALOHA** and **Joint Embedding** model implementations at different *FPRs* (False Positive Rates). More importantly, it provided better results at $FPR = 1\%$ than the other two models.

These results prove the effectiveness of introducing the *Malicious/Benign label* prediction and *Vendor Count* estimation tasks, coming from the **ALOHA** model implementation, as additional targets in order to improve over the **Joint Embedding** model implementation on the *SMART tag* prediction task results.

7.3 Family Prediction and Ranking Evaluation

This section shows the different models' performance evaluation results at the **Malware Family Prediction and Ranking** tasks. More specifically, figures 7.7, 7.8 and 7.9, present, respectively, the models' (mean and standard deviation) accuracy and (*micro* and *macro* averaged) AUC ROC trends obtained by varying the number of anchor samples used during the evaluation. Table 7.7, on the other hand shows the models' scores - while figures 7.10b, 7.10a, 7.10c and 7.10d show the confusion matrixes corresponding to the best predictions - when using the number **k** of anchor samples which produced the overall best accuracy, resulting from the evaluation on the Malware Family Prediction task. Finally, table 7.8 shows the mean and standard deviation of the **MRR** (Mean Reciprocal Rank) and **MAP** (Mean Average Precision) scores corresponding to the produced rankings of the different model implementations.

In particular, the *Joint Embedding* and *Proposed Model* implementations' learned joint embedding representations of PE files (of size 32) were compared also to the ALOHA and ALOHA (M/B only) model implementations' implicit latent representation (of size 512) after the base shared topology. In fact, it is possible to compare the results of the different models (including both ALOHA model implementations) on the family prediction and ranking tasks, given that also the implicit latent representations of the ALOHA-based implementations can be used to compute samples distances. However, when comparing the different scores also the difference in size of the chosen representations should be take into consideration. A smaller representation has, in fact, much higher value than a bigger one, provided it performs well enough, since it allows quicker comparisons between files consequently making the tasks of storing, indexing and querying large databases of malware more efficient.

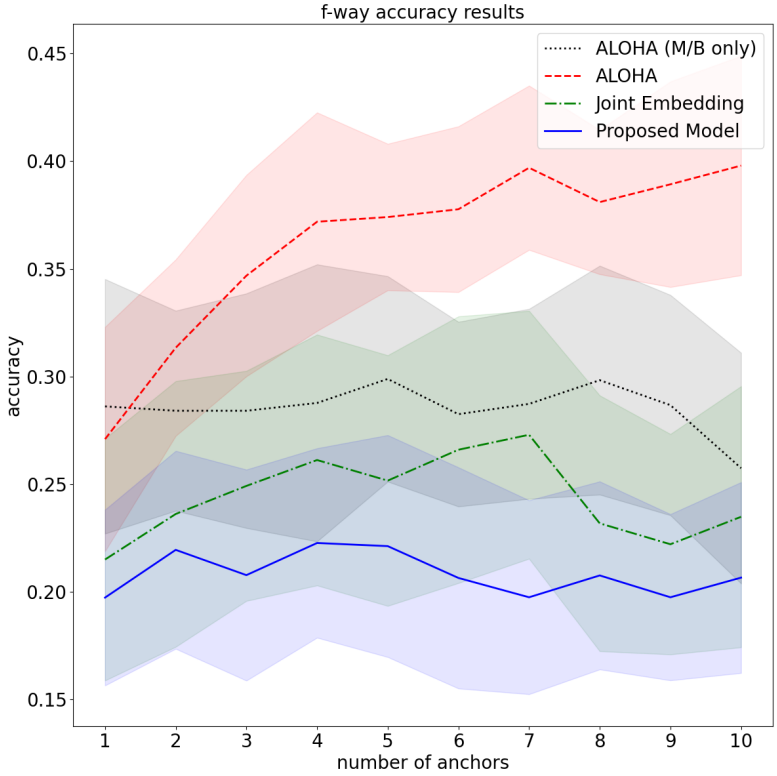


Figure 7.7: **Accuracy Trend** (varying the number of anchor samples used) resulting from the evaluation of the different models on the Malware Family Prediction task. The line represents the *mean* Accuracy, while the shaded region represents the *standard deviation*. Statistics were computed over 15 evaluation of the same models with different query and anchor samples.

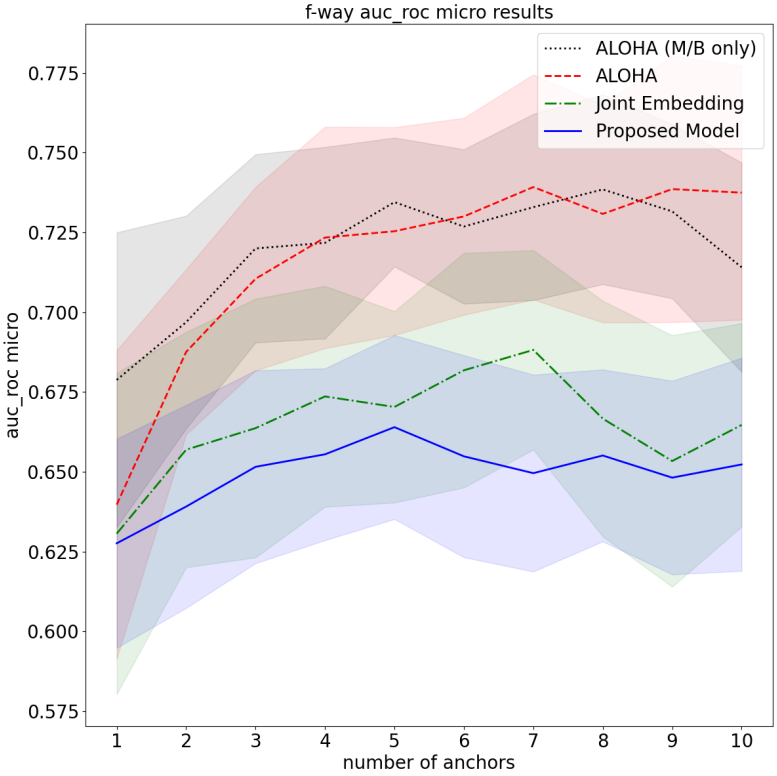


Figure 7.8: **AUC-ROC (Micro) Trend** (varying the number of anchor samples used) resulting from the evaluation of the different models on the Malware Family Prediction task. The line represents the *mean* AUC-ROC (Micro), while the shaded region represents the *standard deviation*. Statistics were computed over 15 evaluation of the same models with different query and anchor samples.

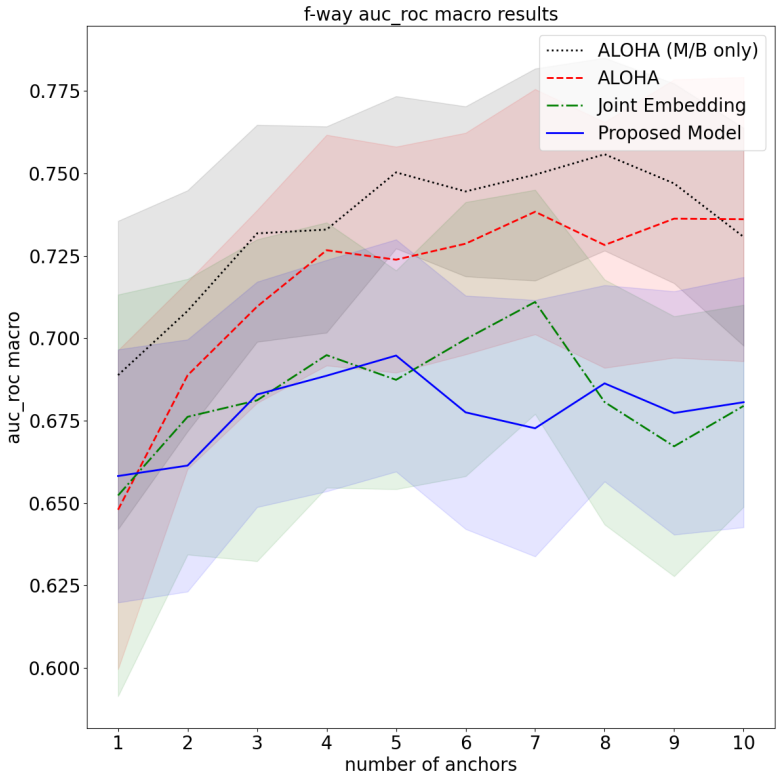
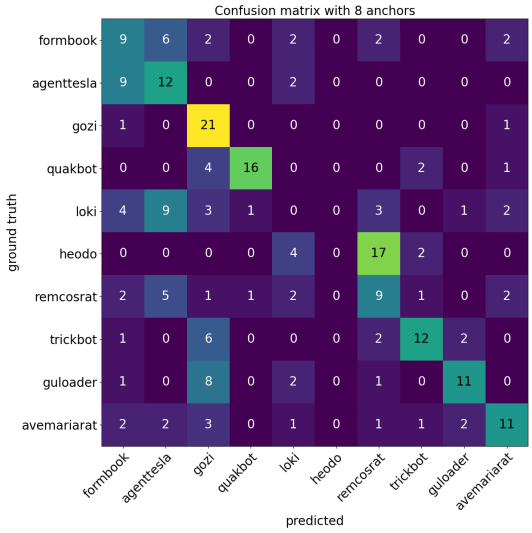


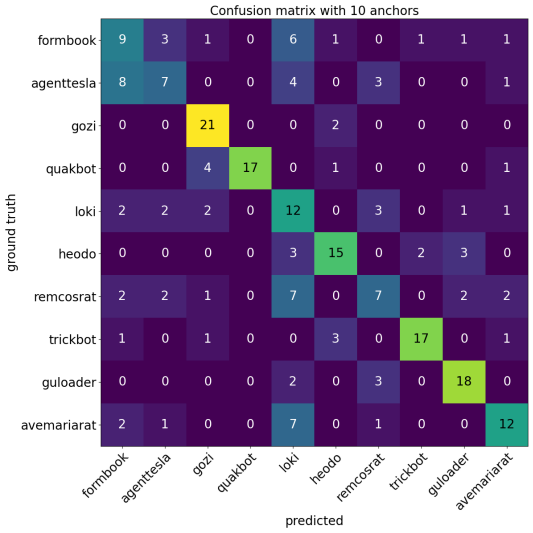
Figure 7.9: **AUC-ROC (Macro) Trend** (varying the number of anchor samples used) resulting from the evaluation of the different models on the Malware Family Prediction task. The line represents the *mean* AUC-ROC (Macro), while the shaded region represents the *standard deviation*. Statistics were computed over 15 evaluation of the same models with different query and anchor samples.

	ALOHA (M/B only)	ALOHA	Joint Embedding	Proposed Model
Accuracy	0.299±0.048	**0.398±0.051**	0.273±0.058	0.223±0.044
Recall Micro	0.299±0.048	**0.398±0.051**	0.273±0.058	0.223±0.044
Recall Macro	0.299±0.048	**0.398±0.051**	0.273±0.058	0.223±0.044
Precision Micro	0.299±0.048	**0.398±0.051**	0.273±0.058	0.223±0.044
Precision Macro	0.295±0.093	**0.465±0.056**	0.291±0.088	0.154±0.055
f1 score Micro	0.299±0.048	**0.398±0.051**	0.273±0.058	0.223±0.044
f1 score Macro	0.248±0.050	**0.386±0.054**	0.216±0.062	0.147±0.042
AUC ROC Micro	0.734±0.020	**0.737±0.040**	0.688±0.031	0.655±0.027
AUC ROC Macro	**0.750±0.023**	0.736±0.043	0.711±0.034	0.689±0.035
n. of anchors	5	10	7	4

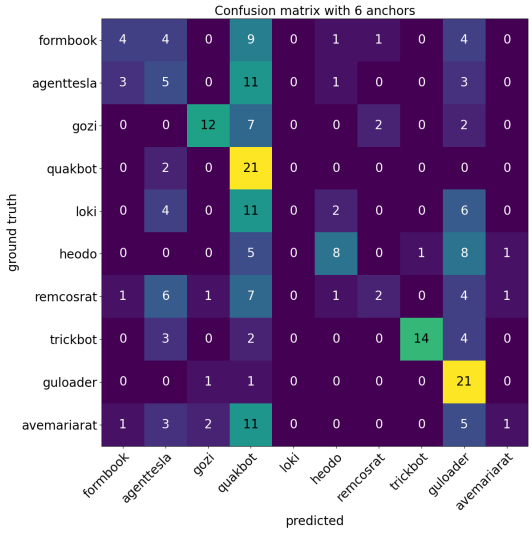
Table 7.7: Mean and standard deviation scores obtained by evaluating the different models on the Malware Family Prediction task with the number k of anchors which produced the maximum accuracy (for each). Statistics were computed over 15 evaluation of the same models with different query and anchor samples. Best results are shown in **bold**.



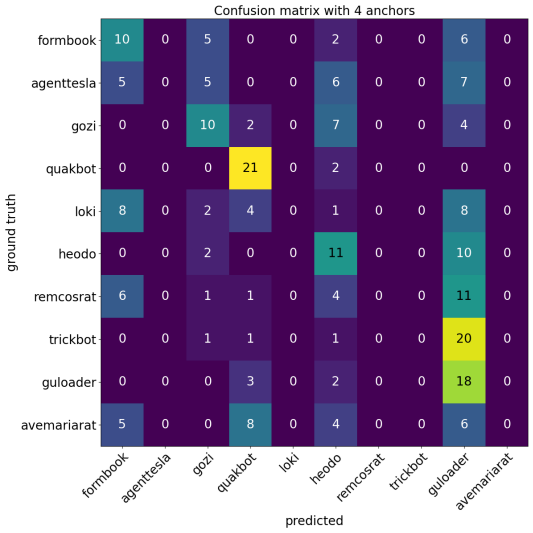
(a) **ALOHA (M/B only)** implementation



(b) **ALOHA** implementation



(c) **Joint Embedding** implementation



(d) **Proposed Model** implementation

Figure 7.10: **Confusion Matrix** corresponding to the best prediction when using the number k of anchors which produced the overall best accuracy, resulting from the evaluation of the different models on the Malware Family Prediction task.

	ALOHA (M/B only)	ALOHA	Joint Embedding	Proposed Model
MRR	0.254±0.015	**0.259±0.008**	0.253±0.012	0.253±0.010
MAP	0.106±0.001	**0.106±0.000**	**0.106±0.000**	**0.106±0.000**

Table 7.8: Mean and standard deviation MRR (Mean Reciprocal Rank) and MAP (Mean Average Precision) results for the different models on the Family Ranking task. Results were aggregated over **2** training runs with different weight initializations and minibatch orderings. Best results are shown in **bold**.

7.3.1 Example rankings

Here are presented, for each model, 4 examples rankings which produced the maximum and minimum AP (Average Precision) and RR (Reciprocal Rank) scores.

Max AP	ALOHA (M/B)			ALOHA		
	Sha256	Label	Family	Sha256	Label	Family
Query	24133d..	0	formbook	e3639a..	0	formbook
0	8fea1e..	3	quakbot	**6f16d3**..	**0**	**formbook**
1	**02ad1e**..	**0**	**formbook**	**0655ee**..	**0**	**formbook**
2	**8da806**..	**0**	**formbook**	**40f8cf**..	**0**	**formbook**
3	0c95f0..	8	guloader	**f7c1ec**..	**0**	**formbook**
4	95fa71..	9	avemariarat	620c36..	2	gozi
5	4b8466..	6	remcosrat	38e003..	6	remcosrat
6	**191008**..	**0**	**formbook**	**93e548**..	**0**	**formbook**
7	768da5..	9	avemariarat	1ddaae..	4	loki
8	**e92575**..	**0**	**formbook**	5b9f32..	3	quakbot
9	f0cae3..	3	quakbot	83d036..	4	loki
Max AP	0.186			0.174		
Next	position 14			position 10		

Max AP	Joint Embedding			Proposed Model		
	Sha256	Label	Family	Sha256	Label	Family
Query	c0a92a..	6	remcosrat	389ec4..	7	trickbot
0	**21f9ef**..	**6**	**remcosrat**	**aa447e**..	**7**	**trickbot**
1	**49c7a5**..	**6**	**remcosrat**	**cbe743**..	**7**	**trickbot**
2	f75cb4..	2	gozi	8dc3da..	6	remcosrat
3	**91b004**..	**6**	**remcosrat**	**e37438**..	**7**	**trickbot**
4	2d54d3..	3	quakbot	**2cadf2**..	**7**	**trickbot**
5	f86765..	9	avemariarat	545bc3..	8	guloader
6	0ea373..	0	formbook	**b56a1e**..	**7**	**trickbot**
7	**4a9558**..	**6**	**remcosrat**	e14108..	6	remcosrat
8	019f79..	7	trickbot	80e880..	2	gozi
9	a2f8c1..	9	avemariarat	**4688b8**..	**7**	**trickbot**
Max AP	0.190			0.183		
Next	position 11			position 17		

Table 7.9: **ALOHA (M/B) Embedding**, **ALOHA**, **Joint Embedding** and **Proposed Model** example rankings (limited to the first 10 samples) having the maximum Average Precision (max AP). The elements matching the query sample are shown in **bold**. The last line of each table indicates the position in the ranking where to find the next element belonging to the same family as the query sample.

Max RR	ALOHA (M/B)			ALOHA		
	Sha256	Label	Family	Sha256	Label	Family
Query	228f0e..	0	formbook	bcec95..	0	formbook
0	**f51577**..	**0**	**formbook**	**7b30aa**..	**0**	**formbook**
1	d7ece5..	3	quakbot	2edfad..	8	guloader
2	040062..	4	loki	ecdf7d..	9	avemariarat
3	80b83b..	1	agenttesla	74fb06..	3	quakbot
4	add45f..	6	remcosrat	0fea0b..	7	trickbot
5	608a6d..	8	guloader	3d4f29..	3	quakbot
6	3cb853..	6	remcosrat	b73730..	7	trickbot
7	fb120a..	6	remcosrat	acf60e..	4	loki
8	60984e..	1	agenttesla	a60e97..	4	loki
9	457344..	6	remcosrat	3b2629..	7	trickbot
Max RR	1.000			1.000		
Next	position 10			position 11		

Max RR	Joint Embedding			Proposed Model		
	Sha256	Label	Family	Sha256	Label	Family
Query	32897c..	2	gozi	807414..	1	agenttesla
0	**b4e412**..	**2**	**gozi**	**d6c10d**..	**1**	**agenttesla**
1	0dac03..	7	trickbot	d0dbb5..	7	trickbot
2	300ec7..	7	trickbot	876680..	2	gozi
3	d9afac..	7	trickbot	6dba26..	7	trickbot
4	b1a303..	0	formbook	bd91fc..	8	guloader
5	b71dc1..	7	trickbot	a31584..	6	remcosrat
6	774cff..	3	quakbot	df907a..	5	heodo
7	1c8aaa..	1	agenttesla	3763ee..	9	avemariarat
8	a6d3dc..	1	agenttesla	277238..	0	formbook
9	d01615..	1	agenttesla	3ad3d1..	5	heodo
Max RR	1.000			1.000		
Next	position 12			position 14		

Table 7.10: **ALOHA (M/B) Embedding**, **ALOHA**, **Joint Embedding** and **Proposed Model** example rankings (limited to the first 10 samples) having the maximum Reciprocal Rank (max RR). The elements matching the query sample are shown in **bold**. The last line of each table indicates the position in the ranking where to find the next element belonging to the same family as the query sample.

Min AP	ALOHA (M/B)			ALOHA		
	Sha256	Label	Family	Sha256	Label	Family
Query	f9b156..	2	gozi	ef89d3..	8	guloader
0	c7fc61..	6	remcosrat	246a1e..	3	quakbot
1	a5ebcb..	5	heodo	4a9e32..	5	heodo
2	494519..	5	heodo	84353f..	5	heodo
3	4469b2..	3	quakbot	f0125d..	2	gozi
4	e0c023..	0	formbook	e0e601..	1	agenttesla
5	a3f0b1..	7	trickbot	0e20f3..	4	loki
6	c27598..	5	heodo	357ab6..	6	remcosrat
7	830072..	7	trickbot	f39db6..	9	avemariarat
8	ca90a8..	6	remcosrat	a98b59..	5	heodo
9	341186..	7	trickbot	719219..	7	trickbot
Min AP	0.062			0.062		
Next	position 20			position 39		

Min AP	Joint Embedding			Proposed Model		
	Sha256	Label	Family	Sha256	Label	Family
Query	c89a67..	7	trickbot	e619c7..	3	quakbot
0	d99c4c..	5	heodo	71cb97..	0	formbook
1	b34153..	5	heodo	10f1ad..	1	agenttesla
2	868b30..	2	gozi	1f1f38..	2	gozi
3	3a5181..	8	guloader	74499f..	1	agenttesla
4	6cf9e4..	4	loki	d75380..	6	remcosrat
5	cf36fa..	8	guloader	d0f6f2..	0	formbook
6	f2a200..	2	gozi	7b30aa..	0	formbook
7	c8eaf6..	0	formbook	0d0364..	6	remcosrat
8	e2c837..	2	gozi	ac0afc..	6	remcosrat
9	1331f9..	0	formbook	d3d0a5..	0	formbook
Min AP	0.063			0.065		
Next	position 67			position 49		

Table 7.11: **ALOHA (M/B) Embedding**, **ALOHA**, **Joint Embedding** and **Proposed Model** example rankings (limited to the first 10 samples) having the minimum Average Precision (max AP). The elements matching the query sample are shown in **bold**. The last line of each table indicates the position in the ranking where to find the next element belonging to the same family as the query sample.

Min RR	ALOHA (M/B)			ALOHA		
	Sha256	Label	Family	Sha256	Label	Family
Query	5f9b8c..	8	guloader	b845b6..	1	agenttesla
0	90770a..	4	loki	6c3efe..	5	heodo
1	f70695..	5	heodo	a09ad5..	0	formbook
2	1a778e..	5	heodo	bad7c7..	2	gozi
3	b0ea42..	4	loki	18498c..	7	trickbot
4	0d77b2..	9	avemariarat	b840f4..	0	formbook
5	300dd4..	0	formbook	3bfc15..	6	remcosrat
6	a98ba3..	0	formbook	76cf64..	6	remcosrat
7	cb5547..	6	remcosrat	8b6690..	9	avemariarat
8	86a419..	9	avemariarat	31cce8..	8	guloader
9	3c2770..	2	gozi	c15a76..	2	gozi
Min RR	0.008			0.009		
Next	position 128			position 113		

Min RR	Joint Embedding			Proposed Model		
	Sha256	Label	Family	Sha256	Label	Family
Query	4a9e32..	5	heodo	dbfdd9..	8	guloader
0	cedbc9..	7	trickbot	861dd1..	2	gozi
1	b48f0a..	6	remcosrat	dca295..	3	quakbot
2	174bce..	2	gozi	c89194..	1	agenttesla
3	8af585..	9	avemariarat	feb0ce..	1	agenttesla
4	4e8a4f..	3	quakbot	26ee0a..	1	agenttesla
5	0558ff..	9	avemariarat	32e56e..	2	gozi
6	7f3487..	0	formbook	510e00..	3	quakbot
7	a6e4d5..	8	guloader	d21f7c..	6	remcosrat
8	850c25..	1	agenttesla	c3df57..	0	formbook
9	853d14..	6	remcosrat	27bc44..	3	quakbot
Min RR	0.011			0.009		
Next	position 94			position 106		

Table 7.12: **ALOHA (M/B) Embedding**, **ALOHA**, **Joint Embedding** and **Proposed Model** example rankings (limited to the first 10 samples) having the minimum Reciprocal Rank (max RR). The elements matching the query sample are shown in **bold**. The last line of each table indicates the position in the ranking where to find the next element belonging to the same family as the query sample.

7.3.2 Comments

Figures 7.7, 7.8 and 7.9 and table 7.7 show that the **Proposed Model**'s learned representation of PE files in the joint embedding space (of size 32) performed worse than the **Joint Embedding** model implementation's one (again of size 32) at the f-way malware family prediction task. Moreover, both the **Proposed Model** and **Joint Embedding** model implementations' learned embeddings performed worse than the **ALOHA (M/B only)** and **ALOHA** models' implicit latent representations - outputted by the base shared topology - (of size 512) at the same task. Figures 7.10b, 7.10a, 7.10c and 7.10d, which show the confusion matrixes corresponding to the best predictions, reflect these results by showing a better confusion matrix for the **ALOHA** model implementation and worse confusion matrixes for both the **Proposed Model** and **Joint Embedding** model implementations. Again, it is to be reminded that the representations of PE files in the joint embedding space learned by these last two models (**Proposed Model** and **Joint Embedding**), of size = 32, are 16 times smaller than the implicit latent representations of the **ALOHA** and **ALOHA (M/B only)** models, of size = 512. Ideally, these smaller representations of PE files should encode the most important characteristics of input samples to be used for the *f-way* family prediction task; however, they were not constructed for such purpose, but rather for assigning a set of SMART tags to each sample in order to describe its functionality. Furthermore, the fact that some families may share the same set of tags can further hinder the generation of a representation in the joint embedding space that can be used to distinguish between such families. Consequently, in this case, the bigger implicit latent representation provided by the **ALOHA** model, being more generic does a better job at encoding those characteristics resulting in higher scores on the *f-way* family prediction task.

Table 7.8, on the other hand, shows that the **Proposed Model** and **Joint Embedding** model's learned representations of PE files in the joint embedding space provide similar **MRR** (Mean Reciprocal Rank) and **MAP** (Mean Average Precision) scores in the Family Ranking task. Again, the **ALOHA** model implicit latent representation - outputted by the base shared topology - performs generally better in this task than the other models but the scores are comparable. In particular, the **MAP** scores of the different models are practically the same, while the **MRR** of the **ALOHA** model is slightly higher than the others. This is reflected by the example rankings shown in tables 7.9, 7.10, 7.11 and 7.12.

These results underline the inadequacy of the **Proposed Model** learned representation of PE files in the joint embedding space when used for the Malware Family Prediction and Ranking tasks, thus highlighting the need for an alternative solution specifically designed for the Malware Family Classification task.

7.4 Family Classification Results

In this section are presented and compared the different Family Classifier model implementations' results. In particular, table 7.13 presents the mean and standard deviation of the classification accuracy obtained by evaluating the Family Classifiers built on top of the **Joint Embedding** and **Proposed Model** implementations along with a Family Classifier model with no Transfer Learning applied (**Family Classifier Only**). Table 7.14, on the other hand, shows the '**Micro**' averaged, '**Macro**' averaged and '**Weighted**' averaged multi-class classification scores resulting from the evaluation of the different Family Classifiers. Finally, figures 7.11, 7.12 and 7.13 show the models' resulting confusion matrixes.

	Joint Embedding Family Classifier	Proposed Model Family Classifier	Family Classifier only
Accuracy	0.445±0.005	**0.465±0.003**	0.414±0.007

Table 7.13: Mean and Standard deviation of the Family Classifier accuracy obtained by evaluating the Family Classifiers built on top of the **Joint Embedding** and **Proposed Model** implementations along with a Family Classifier model with no Transfer Learning applied (**Family Classifier Only**). Results were aggregated over **2** training runs with different weight initializations and minibatch orderings. Best results are shown in **bold**.

		Joint Embedding Family Classifier	Proposed Model Family Classifier	Family Classifier only
Jaccard Similarity	Micro	0.286±0.004	**0.303±0.002**	0.261±0.006
	Macro	0.318±0.001	**0.339±0.002**	0.282±0.011
	Weighted	0.318±0.001	**0.339±0.002**	0.282±0.011
Recall	Micro	0.445±0.005	**0.465±0.003**	0.414±0.007
	Macro	0.445±0.005	**0.465±0.003**	0.413±0.008
	Weighted	0.445±0.005	**0.465±0.003**	0.414±0.007
Precision	Micro	0.445±0.005	**0.465±0.003**	0.414±0.007
	Macro	0.507±0.001	**0.538±0.006**	0.473±0.009
	Weighted	0.507±0.001	**0.538±0.006**	0.473±0.009
F1-score	Micro	0.445±0.005	**0.465±0.003**	0.413±0.008
	Macro	0.438±0.003	**0.460±0.005**	0.397±0.010
	Weighted	0.438±0.003	**0.460±0.005**	0.397±0.010
AUC ROC OVO	Micro	-	-	-
	Macro	0.815±0.004	**0.830±0.002**	0.800±0.012
	Weighted	0.815±0.004	**0.830±0.002**	0.800±0.012
AUC ROC OVR	Micro	-	-	-
	Macro	0.815±0.004	**0.830±0.002**	0.800±0.012
	Weighted	0.815±0.004	**0.830±0.002**	0.800±0.015

Table 7.14: Mean and Standard deviation of the Family Classifier scores obtained by evaluating the Family Classifiers built on top of the **Joint Embedding** and **Proposed Model** implementations along with a Family Classifier model with no Transfer Learning applied (**Family Classifier Only**). Results were aggregated over **2** training runs with different weight initializations and minibatch orderings. Best results are shown in **bold**.

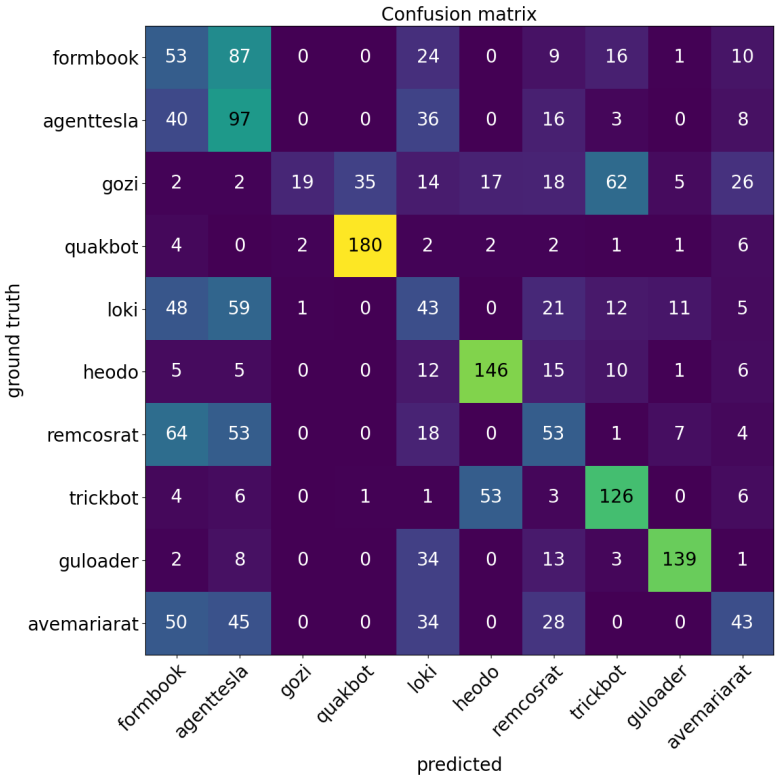


Figure 7.11: Confusion Matrix resulting from the evaluation of the Family Classifier built on top of the Joint Embedding model implementation.

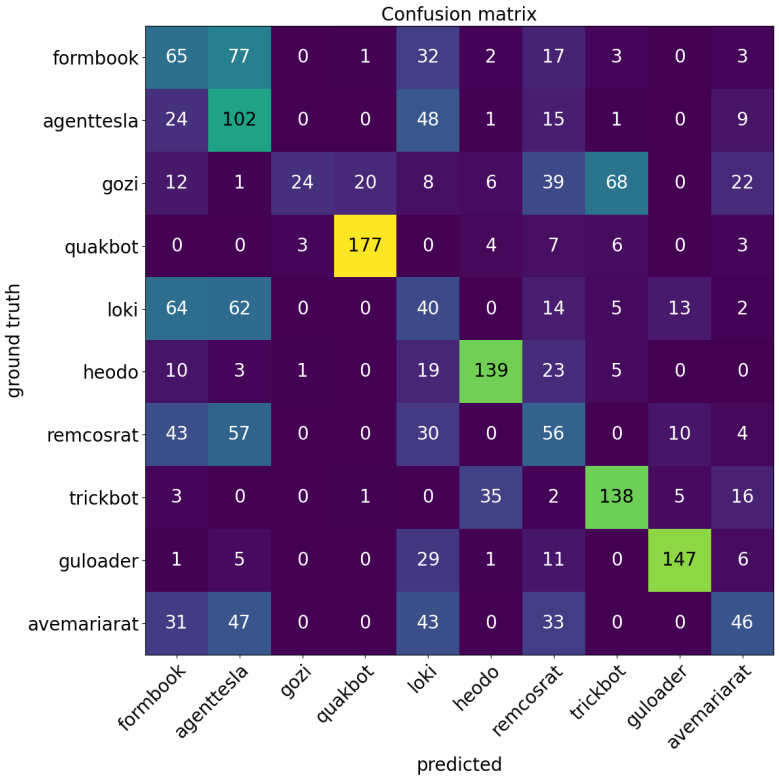


Figure 7.12: Confusion Matrix resulting from the evaluation of the Family Classifier built on top of the Proposed Model implementation.

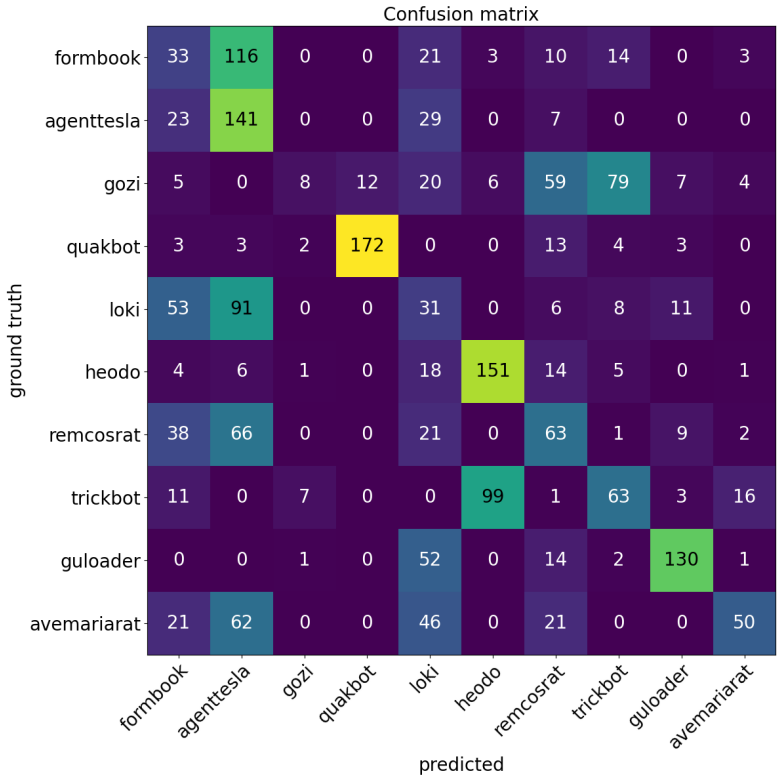


Figure 7.13: Confusion Matrix resulting from the evaluation of the Family Classifier only Model implementation (with no transfer learning).

7.4.1 Comments

As shown in table 7.13, the **Family Classifier** built on top of the **Proposed Model** - by transferring the learned weights of the base shared topology at **Family Classifier** definition effectively re-utilizing the knowledge of a previous **Proposed Model** training run on the Sorel20M dataset, using *transfer learning* - provided a higher *Accuracy* than both the **Family Classifier** built on top of the **Joint Embedding** model implementation and the equally complex **Family Classifier** with no transfer learning applied.

Table 7.14 confirms the better performance of the **Family Classifier** built on top of the **Proposed Model** in the *Malware Family Classification* task by showing that it consistently provided the best '*Micro*', '*Macro*' and '*Weighted*' averaged scores for all multi-class classification statistics used as performance metrics. It is interesting to notice that the **Family Classifier** built on top of the **Joint Embedding** model was, instead, always the second best, while the **Family Classifier** with no transfer learning applied was always last. This proves the effectiveness of utilizing the technique of *transfer learning* to reduce overfitting when training the classifier using the 'small' training subset of the Fresh Dataset.

The confusion matrixes corresponding to the different **Family Classifiers**, shown by figures 7.11, 7.12 and 7.13, reflect these results.

7.5 Computation Time

This section presents an overview of the time needed to complete the experiments conducted for this project using a **Colab Pro** instance (which provides a NVIDIA T4 or P100 GPU, 2 CPU cores, 25GB of RAM and 147GB of disk space). In particular, table 7.15 shows how much time was necessary to complete each step of a single experiment along with the computation time needed for the whole workflow. On the other hand, table 7.16 shows the total computation time spent to complete all the experiments.

	Time Required per Run	N. of Runs per Experiment	Time Required per Experiment
Base Model Training	∼ 3h 5m 34s	3	∼ 9h 16m 42s
Base Model Evaluation	∼ 4m 57s	3	∼ 14m 51s
Base Model Results Computation	∼ 5m 0s	3	∼ 15m 0s
Base Model Evaluation on Fresh Dataset	∼ 8m 12s	3	∼ 24m 36s
Fresh Results Computation	∼ 11s	3	∼ 33s
Family Classifier Training	∼ 31s	3	∼ 1m 33s
Family Classifier Evaluation	∼ 6s	3	∼ 18s
Family Classifier Results Computation	∼ 4s	3	∼ 12s
Mean Results Computation	-	1	∼ 5m 24s
Complete Workflow	∼ 3h 24m 35s	-	∼ 10h 19m 9s

Table 7.15: Average computation time required to complete one experiment.

Total Number of Experiments	Total Computation Time
∼ 90	∼ 928h 43m 30s

Table 7.16: Overall Computation time required to complete all experiments conducted for this project.

As it can be seen in table 7.15, a base model (*ALOHA (M/B only) ALOHA*, *Joint Embedding* or *Proposed Model*) implementation training run took, on average, a little over 3 hours to complete when using generator *alt3* and considering only the first $6M$ training samples of the Sorel20M dataset for 10 epochs. The base model performance evaluation step and the computation of the corresponding results took each 5 minutes (per run). The evaluation of the base model on the Fresh Dataset took instead approximately 8 minutes, on average, while the family classifier training, evaluation and results computation steps required less than 1 minute each (per run). A single workflow cycle thus took approximately 3 hours and a half to complete. By multiplying each step's computation time by the number of runs executed for each experiment and then aggregating the results together considering also the 5 minutes and a half it takes, on average, to compute the model's mean results, it is possible to obtain the approximated overall time spent to complete one whole experiment. In particular, table 7.15 shows that in order to complete one full experiment the **Colab Pro** instance approximately takes, on average, 10 hours and 20 minutes.

The experiments conducted for this project were approximately 90, as reported in the table 7.16. Thereafter, the overall time spent for completing all the experiments was of approximately 928 hours and 43 minutes, which corresponds to ∼ 38 days, 16 hours and 43 minutes of continuous computation.

Chapter 8

Conclusions

This thesis' objective was the creation of an automatic malware signature generation tool capable of generating (implicit or explicit) signatures of PE malicious files with high recall at low false positive rates, possibly capable of detecting not only known malware strains but also novel variants, while also being less susceptible to obfuscation attempts. To achieve this goal an attempt was made by first designing the **Proposed Model**, described in 6.1.1, which improved the results of the **ALOHA** model implementation 5.1.1 (from work [2]) at the Malicious/Benign label prediction - also called Malware detection - task while also providing slightly better results than the **Joint Embedding** model implementation 5.1.2 (from work [3]) at the malware description via SMART tags task. The learned representation of PE files in the Joint Embedding space, which is used to classify a sample as malicious or benign and to assign it the corresponding tags based on its similarity (or distance) to the tag labels in the same space, can in fact be thought as the **implicit** signature of the malicious samples seen during training. These signatures, however, have little interpretability. It is in fact difficult to understand how the model decides to map a file to a specific location in the Joint Embedding space, therefore assigning it a specific set of tags. Moreover, when tested on the *Family Prediction* and *Family Ranking* tasks, these signatures performed poorly. This, however, can be due to the relatively small size of the dataset (half of the Sorel-20M dataset) used during training and to the code approximations introduced in the dataset loading process to speed up the code. In fact, even the **Joint Embedding** model implementation - which provided good results, as stated in paper [3], when trained on a considerably larger number of samples (70M samples) and tested on samples much more similar to the training ones (the authors used the test dataset of the joint embedding model also for the f-way family classification task; such dataset contains samples recorded only a few months later than those used for training) - performed poorly when trained with the Sorel-20M dataset and tested on the *Family Prediction* and *Family Ranking* tasks using the Fresh Dataset (which consists of samples obtained in mid 2021 while the SOREL20M dataset used for training contains samples dating back to 2018 - 2019). The introduction of a **Family Classifier**, specifically trained for the malware family classification task, on top of the **Proposed Model** PE embedding layer provided more meaningful results, although not exceptional, in the family prediction task while also demonstrating the potential of using Transfer learning in this context.

8.1 Future Work

The **Proposed Model** and **Family Classifier** model implementations' have the potential to be very useful tools to the IT-Security field in the current scenario. Indeed, they can be used to detect, describe (via semantic attribute tags) and classify malicious software in an infected system. This is particularly useful for countering a malware infection on a system/network, defining a remediation procedure, identifying possible root causes, and evaluating the severity and potential consequences of the attack. Moreover, by providing a more general description of PE files, they should be able to detect not only known malwares but also novel malware strains/variants, providing some degree of proactive detection.

The current implementations of both models, however, have some limitations; namely the relatively poor results in the family classification task and the lack of interpretability of the resulting implicit signatures. Future works capable of overcoming these shortcomings may be extremely helpful to malware analysts, antivirus software developers and system administrators and could even enable the generation of explicit (and thus more interpretable) signatures derived from the learned implicit ones.

In particular, here are listed some attempts that can be made for overcoming the first limitation:

- Training the **Proposed Model** on the entire Sorel20M dataset (and not just a sub-part of it) with no code optimizations/approximations on a more powerful instance - which can be done with little modifications to the code developed for this project - alone should provide improved results in both the Malware detection and description (via SMART tags) tasks. Moreover, this could also improve the performance of the resulting learned representation of PE Files on the Family prediction and ranking tasks.
- A higher-quality, larger and more recent dataset of PE samples with respect to Sorel20M dataset, could further enhance the performance of the resulting model in all three tasks (especially for the family prediction and ranking tasks). Moreover, a dataset providing also the information regarding the family each sample belongs to could remove the need for a specially crafted Fresh Dataset. This could be beneficial given that the Fresh Dataset was built by downloading samples from Malware Bazaar, which, being a Website where analysts can share malware samples with the InfoSec community, could provide misclassified samples.
- On the other hand, a higher-quality (and possibly larger) dataset of PE malicious files with the corresponding family label with respect to the current Fresh Dataset could improve the performance of the **Family Classifier** model implementation.
- Finally, a model designed following the idea of paper "**Learning from Context: Exploiting and Interpreting File Path Information for Better Malware Detection**" [45] - described in section 3.6.3 - of exploiting additional orthogonal input features deriving statically from the samples has the potential of providing a more informed representation of PE files which may result in better implicit signatures and improved results in all previously mentioned tasks. For example, additional information like the **File Path** where the malicious sample was seen on an endpoint host (as suggested in work [45]), or the image representation of malware samples' byte codes (as described in paper [55]) could be forwarded through a series of convolutional layers to automatically extract the best samples' latent representations to be used in parallel with the currently developed shared base topology output. This, however, implies the need for a dataset providing such additional features.

On the other hand, the fact that the signatures generated with the proposed approach are implicit, and therefore not easily interpretable, can be considered as another limitation. In fact, in order to trust the designed model's decisions malware analysts need to know how they are taken, what aspects of the input samples' features are more relevant in the decision, why some samples are mapped close to each other in the latent space, etc. However, this is difficult - although not impossible - to do when working with Neural Networks. In order to overcome the lack of interpretability of the generated implicit signatures the following attempts could be made:

- Generating a model based on a more interpretable approach, such as **Decision Trees**, with a mapping between inputs and outputs similar to the one of the proposed approach. This is, however, a challenging task which does not guarantee the same performance of the original implicit signatures.
- Extracting explicit signatures, for example in the form of **Yara Rules**, from the implicit ones in one of the following ways:
  - A custom YARA module could be implemented by embedding the proposed **Family Classifier** (or **Proposed Model**, if it becomes good enough at the family prediction task) inside the rule. In particular, the custom module should enable the extraction

of EMBER (version 2.0) feature vectors from the files' binaries and subsequently the generation of the corresponding implicit signatures/latent representations, when the rule is deployed. Then, the generated implicit signatures can be used to decide the family the current sample most probably belongs to between a set of families of interest (the ones the model was trained on). Finally, the rule should be triggered if the predicted family corresponds to the one it represents. This would make the proposed approach compatible with the highly expressive language that is Yara; however, it would not improve the interpretability of the generated signatures since the decision making process is unchanged.

- Alternatively, machine learning model interpretability tools such as **LIME** and **SHAP** could be employed to try understanding the (trained) **Proposed Model** and **Family Classifier** decision making process, highlighting the characteristics of PE samples that influence the decision process the most, for each family. Then, for each family, it could be possible to manually generate one or more Yara Rules by detecting those features that are common to samples of the same family and not for others (through a Union of Disjunctions formulation). This most certainly would help malware analysts in the job of creating Yara rules and can thus be seen as an aiding tool for such purposes. This also has the advantage of easily detecting new malware variants and obfuscated malware, but has the great disadvantage of no longer being an automatic signature generation tool - it is in fact semi-automatic given that a malware analyst has to generate the final Yara Rule.
- Finally, another alternative is to try to use the representation of PE examples in the joint embedding space learned by the **proposed model** to cluster new malware samples into non-overlapping groups and then create, for each group, the Yara Rules manually or through automatic Yara Rules generators such as the aforementioned (in section 3.2.1) **YarGen**, **YaraGenerator**, **Yabin** and **AutoYara**. This could be thought of as a middle ground between the two previous solutions, in the sense that it would generate true Yara Rules while also being (potentially) completely automatic.

Bibliography

[1] "Av-atlas." https://portal.av-atlas.org/, Accessed: 2021-07-19

[2] E. M. Rudd, F. N. Ducau, C. Wild, K. Berlin, and R. Harang, "Aloha: Auxiliary loss optimization for hypothesis augmentation", 2019

[3] F. N. Ducau, E. M. Rudd, T. M. Heppner, A. Long, and K. Berlin, "Automatic malware description via attribute tagging and similarity embedding.", arXiv: Learning, 2019

[4] J. Regan, "What is malware? how malware works and how to prevent it." https://www.avg.com/en/signal/what-is-malware, 2019, Accessed: 2021-03-15

[5] A. P. Namanya, A. Cullen, I. U. Awan, and J. P. Disso, "The world of malware: An overview", 2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud), 2018, pp. 420–427, DOI 10.1109/FiCloud.2018.00067

[6] R. Harang and E. M. Rudd, "Sorel-20m: A large scale benchmark dataset for malicious pe detection", 2020

[7] R. Sharp, "An introduction to malware." https://orbit.dtu.dk/en/publications/an-introduction-to-malware, 2017

[8] R. Moir, "Defining malware: Faq." https://docs.microsoft.com/en-us/previous-versions/tn-archive/dd632948(v=technet.10)?redirectedfrom=MSDN, 2009, Accessed: 2021-03-15

[9] NIST, "malware." https://csrc.nist.gov/glossary/term/malware, Accessed: 2021-03-15

[10] C. Crane, "What is malware? 10 types of malware and how they work." https://www.thesslstore.com/blog/what-is-malware-types-of-malware-how-they-work/, 2020, Accessed: 2021-03-15

[11] Symantec, "Difference between viruses, worms, and trojans." https://knowledge.broadcom.com/external/article?legacyId=TECH98539, 2019, Accessed: 2021-03-15

[12] J. Fruhlinger, "Malware explained: How to prevent, detect and recover from it." https://www.csoonline.com/article/3295877/what-is-malware-viruses-worms-trojans-and-beyond.html, 2019, Accessed: 2021-03-15

[13] P. Mullins, "Malware and its types." http://www.idc-online.com/technical_references/pdfs/information_technology/Malware%20and%20its%20types.pdf, Accessed: 2021-03-15

[14] N. DuPaul, "Common malware types: Cybersecurity 101." https://www.veracode.com/blog/2012/10/common-malware-types-cybersecurity-101, 2012, Accessed: 2021-03-15

[15] S. Ingalls, "Types of malware and best malware protection practices." https://www.esecurityplanet.com/threats/malware-types/, 2021, Accessed: 2021-03-15

[16] MyraSecurity, "What is malware?." https://www.myrasecurity.com/en/what-is-malware/, Accessed: 2021-03-15

[17] K. Baker, "The 11 most common types of malware." https://www.crowdstrike.com/cybersecurity-101/malware/types-of-malware/, 2021, Accessed: 2021-03-15

[18] McAfee, "What is malware?." https://www.mcafee.com/en-us/antivirus/malware.html, Accessed: 2021-03-15

[19] OWASP, "Owasp: Vulnerabilities." https://owasp.org/www-community/vulnerabilities/, Accessed: 2021-07-22

[20] Comtact, "What are the different types of malware?." https://comtact.co.uk/blog/what-are-the-different-types-of-malware, 2019, Accessed: 2021-03-15

[21] P. Szor, "The art of computer virus and defence", Symantec press, 1st ed., 2005, ISBN: 978-0-321-30454-4

[22] A. Sharma and S. K. Sahay, "Evolution and detection of polymorphic and metamorphic malwares: A survey", International Journal of Computer Applications, vol. 90, Mar 2014, pp. 7–11, DOI 10.5120/15544-4098

[23] E. Skoudis and L. Zeltser, "Malware: Fighting malicious code", Prentice Hall Professional, 2004, ISBN: 978-0-131-01405-3

[24] E. Eilam, "Reversing: Secrets of reverse engineering", John Wiley &; Sons, Inc., 2005, ISBN: 9780764574818

[25] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code", Journal in Computer Virology, vol. 2, 08 2006, pp. 67–77, DOI 10.1007/s11416-006-0012-2

[26] M. Sikorski and A. Honig, "Practical malware analysis: The hands-on guide to dissecting malicious software", No Starch Press, 1st ed., 2012, ISBN: 978-1-59327-290-6

[27] L. Sun, S. Versteeg, S. Boztas, and T. Yann, "Pattern recognition techniques for the classification of malware packers", 07 2010, pp. 370–390, DOI 10.1007/978-3-642-14081-5_23

[28] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers", 2015 IEEE Symposium on Security and Privacy, 2015, pp. 659–673, DOI 10.1109/SP.2015.46

[29] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware", IEEE Security Privacy, vol. 6, no. 5, 2008, pp. 65–69, DOI 10.1109/MSP.2008.126

[30] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey." http://pages.cs.wisc.edu/~arinib/writeup.pdf, 2005

[31] I. You and K. Yim, "Malware obfuscation techniques: A brief survey", 11 2010, pp. 297–300, DOI 10.1109/BWCCA.2010.85

[32] E. Konstantinou, "Metamorphic virus: Analysis and detection." https://www.ma.rhul.ac.uk/static/techrep/2008/RHUL-MA-2008-02.pdf, 2008, Technical Report of University of London

[33] I. You and K. Yim, "Malware obfuscation techniques: A brief survey", 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, 2010, pp. 297–300, DOI 10.1109/BWCCA.2010.85

[34] B. Dang, A. Gazet, E. Bachaalany, and S. Josse, "Practical reverse engineering: X86, x64, arm, windows kernel, reversing tools, and obfuscation", Wiley Publishing, 1st ed., 2014, ISBN: 1118787315

[35] R. Perdisci, A. Lanzi, and W. Lee, "Classification of packed executables for accurate computer virus detection", Pattern Recognition Letters, vol. 29, 10 2008, pp. 1941–1946, DOI 10.1016/j.patrec.2008.06.016

[36] V. Nguyen, "A study of polymorphic virus detection", 11 2018, DOI 10.13140/RG.2.2.19853.79842

[37] S. Simon, "What is yara? get to know this malware research tool." https://www.binarydefense.com/what-is-yara-get-to-know-this-malware-research-tool/, Accessed: 2021-03-15

[38] N. Naik, P. Jenkins, R. Cooke, J. Gillett, and Y. Jin, "Evaluating automatically generated yara rules and enhancing their effectiveness", 2020 IEEE Symposium Series on Computational Intelligence (SSCI), 2020, pp. 1146–1153, DOI 10.1109/SSCI47803.2020.9308179

[39] E. Raff, R. Zak, G. Lopez Munoz, W. Fleming, H. S. Anderson, B. Filar, C. Nicholas, and J. Holt, "Automatic yara rule generation using biclustering", Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security, Nov 2020, DOI 10.1145/3411508.3421372

[40] S. Ninja, "Yara: Simple and effective way of dissecting malware." https://resources.infosecinstitute.com/topic/yara-simple-effective-way-dissecting-malware/, Accessed: 2021-03-15

[41] P. Arntz, "Explained: Yara rules." https://blog.malwarebytes.com/security-world/technology/2017/09/explained-yara-rules/#:~:text=YARA%20is%20a%20tool%20that,that%20look%20for%20certain%20characteristics., Accessed: 2021-03-15

[42] Y. Miao, "Understanding heuristic-based scanning vs. sandboxing." https://www.opswat.com/blog/understanding-heuristic-based-scanning-vs-sandboxing, 2015, Accessed: 2021-06-13

[43] Kaspersky, "What is heuristic analysis?." https://usa.kaspersky.com/resource-center/definitions/heuristic-analysis, Accessed: 2021-06-13
[44] Forcepoint, "What is heuristic analysis?." https://www.forcepoint.com/cyber-edu/heuristic-analysis, Accessed: 2021-06-13
[45] A. Kyadige, E. M. Rudd, and K. Berlin, "Learning from context: Exploiting and interpreting file path information for better malware detection", 2019
[46] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling", 09 2016, pp. 230–253, DOI 10.1007/978-3-319-45719-2_11
[47] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features", 2015
[48] H. S. Anderson and P. Roth, "Ember: An open dataset for training static pe malware machine learning models", 2018
[49] H. Chu, "Lightning memory-mapped database manager (lmdb) documentation." http://www.lmdb.tech/doc/, Accessed: 2021-06-22
[50] K. Weinberger, A. Dasgupta, J. Attenberg, J. Langford, and A. Smola, "Feature hashing for large scale multitask learning", 2010
[51] "Lief project." https://github.com/lief-project/LIEF
[52] J. Mu, "Fasttensordataloader." https://discuss.pytorch.org/t/dataloader-much-slower-than-manual-batching/27014/6
[53] CERT-AGID, "Cert-agid threat summary." https://cert-agid.gov.it/tag/riepilogo/
[54] "Malware bazaar." https://bazaar.abuse.ch/browse/
[55] A. Bensaoud, N. Abudawaood, and J. Kalita, "Classifying malware images with convolutional neural network models", 10 2020