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

Summary...

Acknowledgements

Aknowledgments...

Contents

Chapter 1

Introduction

While working on this document, I will mark with the colour red the parts containing drafts and information got from outside sources. I will use the colour orange for parts under active modification, and the colour green for comments (apart from this one).

The accelerating rate of malware incidents on daily basis indicates the magnitude of the problem in malware analysis. While malware analysts detect many malware attacks and incidents, keeping pace with the number and different types of attacks poses a significant challenge to malware analysts. There is no silver bullet with respect to malware, as there is no single malware analysis technique with the capability to treat all malware incidents, as a result analysts select the most suitable malware analysis technique for the specific security incident under consideration [1].

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 [2]:

> *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 as:

> *Malware is a catch-all term to refer to any software designed to cause damage to a single computer, server, or computer network* [3].

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" [4].

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* [4].

Notice that, since the attacker can use a number of different means - such as executable code, interpreted code, scripts, macros etc. - to perpetrate its malicious intents, the term ***software*** should be understood in a broader sense in the above definitions.
Moreover, the computer system whose security policy is attempted to be breached is usually known as the ***target*** for the malware. Instead, the cybercriminal who originally launched the malware with the purpose of attacking one or more targets 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 [2].
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: [5]

- **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**. Nation-state actors (like state-run hacker groups in China and North Korea) and hacker groups such as Anonymous are a perfect example.
- **As a way to entertain themselves**. Some cybercriminals enjoy victimizing others.

Obviously there are also reasons for non-malicious actors to create and/or deploy some types of malware too - for example it can be used to test a system's security.

2.1.2 Malware types

There are numerous 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 or makes the system 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 malware can infect target computers [6]:

- ***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).

  Trojans known as *droppers* are often used to start a worm outbreak, by injecting the worm into users' local networks [7].

  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.

  Trojan horses cannot self-replicate. They rely on the system operators to activate. However, they can grant the attacker remote access permitting him to then perform any malicious activity that is in their interest. Trojan horse programs can affect the host in many different ways, depending on the payload attached to them [8].

- ***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 will lie 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. Viruses are therefore passive. 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 initiate itself the transfer (for example, it may send the infected files as an e-mail attachment) [2].

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***. On the other hand, 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 causing the system to crash. Computer worms may also contain "payloads" to damage the target systems. Payloads are pieces of code written to perform various nefarious actions on the affected computers among which stealing data, deleting files or creating bots - which can lead the infected systems to become part of a botnet [8].

These definitions lead to the observation that viruses require *user intervention* to spread, whereas a worm spreads itself automatically. A virus, however, cannot execute or reproduce unless the application it infected is running. This dependence on a host program makes viruses different from trojans, which require users to download them, and worms, which do not use applications to execute.

Furthermore, 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 [9].

By what they do

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

- ***Adware***. *Adware*, or "Advertising supported software", is any software package which automatically plays, displays, or downloads advertisements to a computer. Some adware may also re-direct the user's browser to dubious websites. These advertisements can be in the form of a pop-up ads or ad banners in websites, or advertisements displayed by software, that lure the user into making a purchase. The goal of Adware is to generate revenue for its author.

  Often times software and application authors offer "free", or discounted, versions of their creations that come bundled with adware. Adware, in fact, is usually seen by the developers as a way to recover development costs. The income derived from ads may motivate the developer to continue developing, maintaining and upgrading his software product. On the other hand users may see advertisements as annoyances, interruptions, or as distractions from the task at hand [7].

  Adware, by itself, is annoying but somewhat harmless, since it is solely designed to deliver ads; however, adware often comes bundled with spyware (such as keyloggers), and/or other privacy-invasive software that is capable of tracking user activity and steal information. Adware-spyware bundles are therefore much more dangerous then adware on its own [10].

- ***Backdoor***. A *backdoor*, also called Remote Access Trojan (RAT), is a vulnerability deliberately buried into software's code that allows to bypass typical protection mechanisms, like credentials-based login authentication. In other words, it is a method of circumventing normal authentication procedures. 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 [7].

Many device or software manufacturers ship their products with intentionally hidden backdoors to allow company personnel or law enforcement to access the system when needed [11]. Alternatively, backdoors are sometimes hidden in programs also by intelligence services. For example, Cisco network routers, which process large volumes of global internet traffic, in the past were equipped with backdoors intended for US Secret Service use [12].

However, when used by malicious actors, backdoors grant access to attackers without the user knowledge, thus putting the system in real danger.

- ***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. For example it can force the browser to send the user to a new search page, slow down the loading, change the victim's home page, install unwanted toolbars, redirect the user to specific sites, and display unwanted ads without the user consent.
  It can be used to make money off ads, to steal information from users, or to infect the systems with other malware by redirecting users to malicious websites [11].

- ***Bots/Botnet***. In general, *bots* (short for 'robots') are software programs designed to automatically perform specific operations. Bots were originally developed to programmatically manage chat IRC channels - Internet Relay Chat: a text-based communication protocol appeared in 1989.
  Some bots are still being 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* [8]. Botnets are frequently used for DDoS (Distributed Denial of Service) attacks, but there are other ways that botnets can be useful to cybercriminals: [5]

  - **Brute force & credential stuffing** - Bots can be used to carry out different types of brute force attacks on websites. For example they can use a pre-configured list of usernames and passwords combinations on website login pages with the hope of finding a winning combination, after enough tries.
  - **Data and content scraping** - Botnets can be used as web spiders to scour websites and databases to gather useful information - such as site content, pricing sheets, etc. - which can be used to obtain an unfair advantage against the competition.
  - **Botnet-as-a-service opportunities** - Botnets are sometimes rented out by their creators to all kinds of malicious users - including less tech-savvy ones. Doing so, even inexperienced attackers can carry out attacks, such as taking down a target's servers and networks with a DDoS, using these mercenary bots. This service model is sometimes called malware-as-a-service.
  - **Spambot** - A botnet can also be used to act as a spambot and render advertisements on websites.
  - **Malware distributor** - Finally Botnets can even be used for distributing malware disguised as popular search items on download sites.

- ***Crypto-miner***. Crypto-miners are a relatively new family of malware. Cybercriminals employ this type of malicious tools to mine Bitcoin and/or other bitcoin-alike digital currencies on the target machine. The victim system's computing power is used for this, without the owner realising it. 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.

  In some cases, the use of crypto-miners may be deemed legal. For example they could be used to monetize websites, granted that the site operator clearly informed visitors of the use of such tools [12].

Finally, according to ESET, most crypto-miners focus mostly on *Monero* as target crypto-currency because it offers anonymous transactions and can be mined with regular CPUs and GPUs instead of expensive, specialized hardware [5].

- ***File-less malware***. File-less malware is a type of memory-resident malware that uses legitimate code already existing within the target computer or device to carry out attacks. As the term suggests, it is malware that operates from a victim's computer memory, not from files on the hard drive, taking advantage of legitimate tools and software (known as "LOLBins" [5]) that already exist within the system. File-less malware attacks leave no malware files to scan and no malicious processes to detect. Since there are no files to scan, it is harder to detect and remove than traditional malware; this makes them up to ten times more successful than traditional malware attacks [13]. Furthermore, it also renders forensics more difficult because when the victim's computer is rebooted the malware disappears.

- ***Keylogger***. Keystroke logging (often called *keylogging*) is the action of secretly tracking (or logging) keystrokes on a keyboard, without the person using the keyboard knowing that its actions are being monitored. The collected information is stored and then sent to the attacker who can then use the data to figure out passwords, usernames and payment details, for example. There are various methods used to perform keylogging, ranging from hardware and software-based approaches to the more sophisticated electromagnetic and acoustic analysis [7]. Key loggers can be inserted into a system through phishing, social engineering or malicious downloads.

  There are various methods used to perform keylogging, ranging from hardware and software based approaches to electromagnetic and acoustic analysis.

  To this extent keyloggers can be considered as a sub-category of spyware.

  Keylogging also has legitimate uses, in fact it is often used by law enforcement, parents, and jealous or suspicious spouses. The most common use, however, is in the workplace, where employers monitor employee use of company computers.

- ***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 access unencrypted credit card numbers [11].

- ***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 requests a ransom from the host/network users. In particular it encrypts data on the infected system (or anyway locks down the system so that the users have no access) and only unblocks it when the correct password - decryption key - is entered. The latter is not given to the victims until after they have paid the ransom to the attacker. Messages informing the system user of the attack and demanding a ransom are usually displayed. Without the correct decryption key, it's mathematically impossible for victims to decrypt and regain access to their files.

  Digital currencies such as Bitcoin and Ether are the most common means of payment, making it difficult to track the cybercriminals. Moreover, paying the ransom does not guarantee the user to receive the necessary decryption key or that the one provided is correct and functions properly. Additionally, some forms of ransomware threaten victims to publicize sensitive information within the encrypted data.

  Ransomware is one of the most profitable, and therefore one of the most popular, and dangerous kinds of malware programs of the past few years.

  The "Five Uneasy E's" of ransomware, according to Tim Femister [14] - vice president of digital infrastructure at ConvergeOne - are:

  - **Exfiltrate**: Capture and send data to a remote attacker server for later leverage.
  - **Eliminate**: Identify and delete enterprise backups to improve odds of payment.
  - **Encrypt**: Use leading encryption protocols to fully encrypt data.
  - **Expose**: Provide proof of data and threaten public exposure and a data auction if payment is not made.

  - **Extort**: Demand an exorbitant payment paid via cryptocurrency.

- ***Rogue Security Software***. *Rogue Security Software* can be considered as a from of scareware. This type of malware program presents itself as a security tool to remove risks from the user's system. In reality, this fake security software installs more malware onto their system [11].

- ***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 remotely execute files, log user activities, access/steal information, modify system configurations, alter software (including security software), install hidden malware, mount attacks on other systems or control the computer as part of a botnet. Since a rootkit operates stealthily and continually hides its presence, its prevention, detection and removal can be difficult; in fact, typical security product are often not effective in detecting rootkits. Rootkit detection therefore often relies on manual methods such as monitoring the computer's behaviour for irregular activity, scanning system file signatures, and analysing storage dumps [10].

  More recently, the term "rootkit" has also often been used to refer to concealment routines in a malicious program. These routines are highly advanced and complex and 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. The same is true when a human attacker directly breaks into a computer. Techniques known as rootkits allow for this concealment by modifying the host's operating system so that malware is hidden from the user. They can prevent a malicious process from being visible in the system's process list or prevent its files from being read [7].

  Traditionally, rootkits can install themselves in kernel level (ring 0), although some sources state that they can install themselves all the way up to user level (ring 3). This means that they can get as much (or as little) access as necessary.

  There are different types of rootkits, which are typically categorized by the reach of the system they affect: [11]

  - ***User-level/application level rootkits*** - User-mode rootkits run in Ring 3, along with other applications as user. They can alter security settings, allowing the attacker to replace executables and system libraries.
  - ***Kernel-level rootkits*** - Kernel-mode rootkits run in ring 0, the highest operating system privileges (Ring 0). They manage to do so by modifying the core functionality of the operating system - the kernel. They usually add code or replace portions of the core operating system, including both the kernel and associated device drivers.
  - ***Bootkit rootkits*** - A Bootkit rootkit is a type of kernel-mode rootkit which infects startup code like the Master Boot Record (MBR), Volume Boot Record (VBR), or boot sector, subverting the kernel upon computer start up.
  - ***Virtualization rootkits*** - This type of rootkit, also called *Hypervisor rootkit*, runs in Ring -1 (before the kernel) and hosts the target operating system as a virtual machine. It manages to do so by exploiting hardware virtualization features. This in turn enables the rootkit to intercept hardware calls made by the original OS.
  - ***Hardware/firmware rootkits*** - A firmware rootkit uses device or platform firmware to create a persistent malware image in hardware. The rootkit hides in firmware, because the latter is not usually inspected for code integrity.

- ***Scareware***. Scareware is a generic term for malware that uses social engineering to frighten and manipulate a user, inducing him into thinking their system is vulnerable or has been attacked. 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 [12].

Both Rogue Security Software and Ransomware can be considered as scareware, together with other scam software.

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 using tricks - such as browser redirect loops - to fool the victim into thinking they have done more damage than they really have [9].

- ***Spyware***.

  Spyware, another name for *privacy-invasive software*, is a type of malicious software that uses functions in the infected host's operating system with the aim of spying on the 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). The presence of spyware is typically hidden from the user, and can be difficult to detect.

  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 and redirecting web browser activity. Spyware is known to change computer settings, often resulting in slow connection speeds, different home pages, and/or loss of Internet connection or functionality of other programs. They spread by attaching themselves to legitimate software, Trojan horses, or even by exploiting known software vulnerabilities [7].

  Law enforcement, government agencies and information security organizations often use spyware to monitor communications in a sensitive environment or during an investigation. Spyware is however also available to private consumers, allowing them to spy on their employees, spouse and children [15].

Other cyber-threats

Other cyber threats which are not strictly 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. Some bugs may be caused by compilers or operating systems component used by the program.

  Minor bugs only slightly affect the behaviour of a program. Therefore it can be a long time before they are discovered. On the other hand, more significant bugs can cause crashes or freezes. It is safe to say that almost all software has bugs and most bugs go unnoticed or have a slight impact on the user.

  However, other bugs qualify as security bugs. These are the most serious type of bugs since they can allow attackers to bypass access controls such as user authentication, override access privileges, or steal data.

  The frequency of bugs can be reduced through developer training, quality control, and code analysis tools [10].

- ***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 an ad on a legitimate website. When a user clicks on the ad, code in the ad either redirects them to a malicious website or installs malware on their computer.

  In some cases, the malware embedded in an ad might 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 commonly used to perform cyber attacks. Particularly in a phishing attack, the attacker attempts, through email messages, to trick users into divulging passwords (or anyway 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" [16].

  There are different types of Phishing. Here are mentioned some of them: [11]

  - *Deceptive Phishing* - The most common type. It uses an email headline with a sense of urgency from a know contact. This attack blends legitimate links with malicious code, modifies brand logos, and evades detection with minimal content.
  - *Spear Phishing* - Spear phishing targets specific users or organizations by researching the victim to maximise trick potential. For example the attacker may explore social media, record out-of-office notifications, compromise API tokens etc. in order to better fool the target user.
  - *Whaling* - Whaling is similar to spear phishing, but even more targeted. In fact, it targets chief officers of organizations using various social engineering tricks such as impersonating employees or co-workers and using phone calls - to name a few - to give a sense of legitimacy to malicious emails.
  - *Vishing* - Vishing targets phone users. It uses the Voice over Internet Protocol (VoIP), technical jargon, and ID spoofing to trick a caller into revealing sensitive information.
  - *Smishing* - Smishing also targets phone users. It uses, however malicious text messages (SMS).
  - *Pharming* - Pharming leverages cache poisoning against the DNS with the objective of redirecting users to fake websites.

- ***Spam***.

  In cybersecurity, unsolicited emails are generally referred to as *spam*. Typically, spam includes emails carrying unsolicited advertisements, fraud attempts, links to malicious websites or malicious attachments. Most spam emails contain one or more of the following: [11]

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

  Spam might be one of the most universally understood forms of malicious attacks. As billions of users enable email for their everyday lives, it makes sense that malicious actors try to sneak into their inbox. Some of the most common types of spam emails include fake responses, PayPal, returned mail, and social media. All of which are disguised as legitimate but contain malware.

General considerations on malware types

Malware samples are usually categorised both by a means of infection and a 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 [9].
Finally, many instances of malware fit into multiple categories: for example Stuxnet is both a worm, a virus and a rootkit.

Furthermore, in recent years, targeted attacks on mobile devices have also become increasingly popular. In fact, among the huge amount of available apps, an increasing quantity is not desirable; the problem is even worse when considering third-party app stores. Even when app store providers impose filters and manual checks to prevent the malicious apps from being available, some inevitably slip through. These mobile malware threats are as various as those targeting desktops and include Trojans, Ransomware, Advertising click fraud and more. They are mostly distributed through phishing and malicious downloads and are a particular problem for jail-broken phones, which tend to lack the default protections that were part of those devices' original operating systems.

2.1.3 Malware History

Malware history began 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 [17]. There are numerous examples of malware created at that time within a laboratory setting: for example the ***Darwing game*** in 1962, ***Creeper*** in 1971, ***Rabbit Virus*** in 1974 and ***Pervading Animal*** in 1975.

In particular, the malware called ***Creeper*** was designed to infect mainframes on ARPANET. The program did not alter the machines' functions, nor it stole or deleted data. It only displayed the message "I'm the creeper: Catch me if you can" while illegitimately spreading from one mainframe to another. This malware was later upgraded with the ability to self-replicate and became the first known computer worm [18].

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.

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. Then, following the success of that prank gone wild, the first Microsoft PC virus, called ***Brain***, was 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 [8]. 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 [17].

In the following decades malware has evolved both regarding its complexity and malware sample numbers.

The growth in malware complexity can be divided 5 different malware generations [8]:

- *First generation*: (DOS Viruses) malware mainly replicate with the assistance of human activity
- *Second generation*: malware self-replicate without help and share the functionality characteristics of the first generation. They propagate through files and media.
- *Third generation*: malware utilise the capabilities of the internet in their propagation vectors leading to big impact viruses.
- *Fourth generation*: malware are more organization-specific and use multiple vectors to attack mainly anti-virus software or systems due to the commercialisation of malware.
- *Fifth generation*: malware is used in cyberwarfare and the now popular malware as-a-service makes its appearance.

Each jump in generation is linked to an increase in malware complexity and more propagation vectors being available. Newer generations of malware always re-utilise older techniques while introducing newer ones. Finally newer generations are more and more evasive due to the commercial value in having access to exploited systems.

In the appendix, section 10.1, a list of the most famous examples of malware attacks/incidents of recent history can be found.

2.2 Detection evasion

From the creation of the first malware in 1970 [19], there has been a strong competition between attackers and defenders. 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.

The first type 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 [20] which, to evade such opcode signatures, employs several obfuscation techniques and can create variants of itself. This posed a challenge to anti-malware developers.

The first time a malware has been recognised to exhibit detection avoidance behaviour was in 1986 with the *Brain* virus [21]. In fact, such malware 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 has become the number one priority: the longer the malware remains undetected, the more harm it can do and the more profitable it is to its writer [8].

2.2.1 Reverse-Engineering

Reverse engineering, in broad terms, indicates the process of extracting knowledge, ideas, design philosophy etc. from anything man-made [22].

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

Reversing is used extensively by both malicious actors and investigators but with opposing purposes. Malware developers often use it to discover vulnerabilities in systems or programs, while analysts and antivirus software developers use it to analyse malicious programs to understand how they work, what damages they can cause, how they infect the system and reproduce, 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 threats associated). This process allows security teams to develop effective detection techniques against the analysed malicious code, contain its damage, reverse its effects on the system, develop 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 design methods to guard systems against future infections [23].

Initially malware analysts/researchers had to manually analyse each malware sample. This process is however complex, requires high expertise, and is time-intensive. Moreover, the number of malware samples that need to be analysed on a daily basis is nowadays of the order of hundreds.

This implies that the analysis of malware samples can no longer be done exclusively manually. 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 [8].

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, being 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 ones generating a more human-interpretable code listing. On the latter, control flow, data flow analysis, and many others static techniques can be employed to try understating the program functionality and inner workings, among other useful information [23].

Static analysis advantages are, among others, that it takes into account the entire program code and it is also usually faster (and safer) than the dynamic one. 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 [8].

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, they 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 [23]. 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 taking a snapshot of the original system state, 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 [8].

When using dynamic techniques, however, malware analysts don't simply run malware executables on their own computer, which most probably is even 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 (and isolated) 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 are techniques originally meant to make the reverse engineering process difficult for a hacker or any malicious user. The main objective of various anti-reverse engineering techniques is simply to complicate the process of reversing as much as possible. For example an attacker could use the disassembly of a binary in order to get an insight of the logic of the code as well as getting hidden information.

Recently anti-reversing techniques are, however, extensively used also by malware authors in order to make their creations difficult to analyse in an attempt to 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. 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 [24]. The attackers' usage of these techniques thus implies a time-consuming analysis for malware analysts, ultimately preventing the retrieval of the source code in a reasonable time.

Any executable code can be reverse engineered, but by armouring their code with anti-disassembly and anti-debugging techniques, attackers increase the skill level required by analysts. 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 inherent weaknesses present in disassembler algorithms. Moreover, disassemblers, in order to work properly, make certain assumptions on the code being analysed. However, when these assumptions are not met, there is an opportunity for malware authors to deceive the analyst.

For example, while disassembling a program, sequences of executable code can have multiple disassembly representations, some of which may be invalid and obscure the real purpose of the program. Thus, the 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 [24].

There are two types of disassembler algorithms: linear and flow-oriented (recursive). The linear one is easier to implement, but it is also more simplistic and error-prone.

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 [24].

Linear disassemblers are easy to implement and work reasonably well when working with small sections of code. They introduce, however, occasional errors even with non-malicious binaries. The main drawback of this technique is that it blindly disassembles code until the end of the section, assuming the data is nothing but instructions packed together, without being able to distinguish 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 will be blindly disassembled and interpreted by the linear disassembler as instructions.

Malware authors can exploit this weakness of linear-disassembly algorithms implanting data bytes that form the opcodes of multi-byte instructions in the code section.

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* [24].

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, if disassembling a code section we find a JMP instruction, this type of disassembler will not blindly parse the bytes immediately following the JMP instruction?s ones, but it will disassemble the bytes at the jump destination address.

This behaviour is more resilient and generally provides better results, 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.
Particularly, 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 first one branch or the other. However, in handwritten assembly code and anti-disassembly code, taking first one branch or the other can often produce different disassembly 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 [24].

Here is an example:

```
74 03   jz  loc
75 01   jnz loc

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, since it disassembles just one instruction at a time, won't recognize this combination as being an unconditional branch. During the disassembly process, in fact, if a **jnz** instruction is encountered, the disassembler will take the false branch of the instruction and will continue 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 [24].

Example:

```
33 C0   xor eax, eax
74 01   jz  loc

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 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. This byte is called *rogue byte* because it is not part of the program logic flow and it is inserted in the code with the only purpose of fooling the disassembler [24].

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. Exploiting these conditions we obtain what are called *impossible disassembly* techniques. The code produced using these techniques can however be disassembled, but only 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 the 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. While the **FF** byte is in reality part of both instructions that actually execute, the disassembler is not able to recognise this.

The 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 be inserted at any location in a program code in order to fool disassemblers. However this sequence it is also easily recognisable by reverse engineers and substituted with **NOP** instructions using IDA Pro or other instruments and/or scripts. Another alternative is to interpret this sequence as data bytes forcing the disassembler to skip it.

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

Obscuring Flow Control

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 [24].

Control-flow analysis can however be easily defeated by malware authors.

The Function Pointer Problem Function pointers are a common programming idiom present in programming languages such as **C**, while being extensively used in the background in object oriented languages like **C++** and **Java** [24].

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. Since, doing so, the function is being invoked indirectly through a variable instead of directly through a fixed identifier or address, such invocation is also known as an "indirect" call. 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 the disassembler about the program control flow. Moreover, if function pointers are used in specially crafted, or non-standard code, the resulting code 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 we already mentioned the *call* and *jmp* instructions, however there are more [24].

The counterpart to the *call* instruction is *retn*. When a call instruction is reached during program execution, a return pointer is pushed on the stack, before jumping to the call instruction target. This return pointer in the stack will point to the address of the instruction immediately following the end of the *call* instruction itself. Therefore a *call* instruction can be seen as the

combination of a *jmp* and *push*; a *retn* instruction, on the other hand, is the combination of *pop* and *jmp*.

The *retn* instruction pops the last value pushed to the stack and jumps to it; it 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. As added benefit the disassembler will also prematurely terminate the function being analysed.

Misusing Structured Exception Handlers Another powerful anti-disassembly technique exploits the Structured Exception Handling (**SEH**) mechanism. Performing program flow control using this mechanism is able to fool both disassemblers and debuggers [24].

SEH provides programs a way to handle error conditions intelligently. *C++* and other programming languages heavily rely on exception handling (and therefore on **SEH**) when compiled for x86 systems.

Exceptions can be triggered for numerous 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 exception, 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 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. A malware using these techniques usually alter its normal control flow paths or causes crashes if it detects it is running in a debugger, thus interfering with analysis [24].

Windows Debugger Detection

In Windows OS various techniques can be used to detect if a process is being run in a debugger: from exploiting the Windows API itself, to manually checking memory structures for debugging artefacts [24].

Using the Windows API One of the most obvious, and simple, 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 debugger detection [24].

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

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

- **IsDebuggerPresent** This is the simplest API function that can be used for debugger detection. 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*). This function 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 will check 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 you want to get. When using the value *ProcessDebugPort* for this parameter, for example, the function will return a zero if the process is not currently being debugged; a non-zero value representing the debugger port number will instead be returned otherwise.

- **OutputDebugString** This function, originally designed to just send a string to a debugger for display, can 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
  RunMaliciousPayload();
}

```

Listing 2.4. OutputDebugString debugger detection

Manually Checking Structures Malware authors usually don't exploit the Windows API functions for detecting the presence of a debugger, but they prefer checking the PEB structure (and others) by themselves. One of the reasons why they usually don't like using Windows API functions is that API calls can be easily hooked by a rootkit to return false information, thus thwarting this technique [24].

- **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.

Malware can 'manually' check the *BeingDebugged* flag within the PEB structure to understand if a debugger is attached its process. More precisely if this flag is zero it means that no debugger is attached.

Example of code listing performing 'manual' *BeingDebugged* check:

```
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 detecting this code sequence in the code and then wither manually changing the *BeingDebugged* flag to zero, or forcing the jump to be taken (or not) by manually 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 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 traces of their presence on the system. Malicious programs can therefore be designed to search for these traces in the system in order to determine when it is being analysed. For example malware can search for references to debuggers in the registry keys [24].

Moreover, malware can also be designed to search the system for files and directories commonly related to debuggers, such as debugger program executables.

Furthermore, malware 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 [24].

INT Scanning A common anti-debugging technique used by malware authors consists in making the process scan its own code in search for an **INT 3** (opcode *0xCC*). **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 [24].

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

Performing Code Checksums Another anti-debugger technique consists in calculating the checksum of a section of the process' own code. This has the same net effects as scanning the code for software interrupts. However, instead of explicitly searching for a specific opcode (*0xCC*) in the process code, this check performs a cyclic redundancy check (CRC) or a MD5 checksum of the malware code [24].

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 [24].

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

1. Recording 2 timestamps before and 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 before and 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.

- **Using the *rdtsc* Instruction** The most common timing check method uses the *rdtsc* instruction. This instruction returns the number of ticks since the last system reboot. Malware authors thus use it as described above: rdtsc is called twice, once before and once after some other operations, and then the difference between the results is calculated. If too much time has elapsed between the two calls it means that a debugger is probably being used.
- **Using *QueryPerformanceCounter* and *GetTickCount*** These are two Windows API functions that can be used similarly to *rdtsc* for debugger detection.

  More precisely *QueryPerformanceCounter* can be called to query a high-resolution counter available to processors which stores counts of activities performed by the processor.

  The function *GetTickCounter*, on the other hand, returns the number of milliseconds that have elapsed since the last reboot, very similarly to what the *rdtsc* instruction does.

  Both of those functions, when used as described above, allow the malware to detect the presence of a debugger.

Anti-debugging though the use of timing checks can be discovered by malware analysts during debugging or static analysis by identifying specific sequences of instructions. Moreover, these checks usually detect debuggers only 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 could avoid setting breakpoints and single-stepping in those regions of code, or 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 [24].

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 kind of artefacts can be specific files, processes, registry keys, services, network device adapters etc. [24].

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 needs to create a MAC address for the virtual machine, to associate to its NIC. However, depending on VMware configuration, this may lead to the network adapter being able to identify VMware usage. VMware utilises, in fact, addresses with a specific starting sequence which depends on its current version. Therefore a malicious software just needs to check the system MAC address against common VMware values.

In order to counter anti-virtual machine techniques, malware analysts need to apply a two step process: identify the check for VMware artefacts and then 'manually' patch it. 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 [24].

However, those instructions previously mentioned are not typically used within a malicious program unless it is specifically performing VMware detection, because they are useless if 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 [24].

When packers are used on malicious programs, the malicious code appearance is changed as a consequence of the compression and/or encryption. The packed file will thus hinder basic static analysis and malware detection. 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 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 [25].

Malware authors have thus 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 generally is, however, not easy.

A packed file usually contains two basic components:

- A number of data blocks containing the compressed and/or encrypted 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 has been fully completed the control flow is transferred, with a *jmp*, *call* or the more stealthy *retn* instruction (also referred to as the *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 [26] identified 6 packer main types 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 only 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 four categories [27]:

- **Compressors** utilise compression 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. No compression is usually done. Malware developers widely use crypters such as Yoda's Crypter and PolyCrypt PE.
- **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 file. These files within the package can then be unpacked and accessed without extracting them to disk. Some common 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 [24].

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* ([24], [27]).

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 to hinder analysis.

Automated dynamic unpackers, instead, 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 program/script to revert it. This process is however 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 [28], both for humans and automatic tools. To do so, it transforms a program into a new (structurally different) and more difficult to analyse version while retaining the same functionality as the original (the new version of the program is said to be *computationally equivalent* to the original one) [29].

Originally, this technology was conceived for legitimate purposes to protect the intellectual property of software developers; however it has been widely exploited by malware authors to evade detection [30]. Particularly, in order to elude anti-malware scanners, malware can, using obfuscation techniques, evolve their body into new generations [31], 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 difficult ways to strengthen the resulting obfuscation [32].

Data-Based Obfuscation

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

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 [32].

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. The use of *fully Homomorphic* mappings (operation-preserving mappings) allow us to perform some operations on the encoded data before decoding it back, thus overcoming the previous technique's flaw. *Fully Homomorphic* mappings are however still not widely used because still too inefficient [32].

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 [32].

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. Exploiting the equivalence between different combinations of different operators the code can be changed arbitrarily without changing the effective program operation and final result [32]. Here are some examples of operators 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 [31].

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 [31].

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 [32].

For example, the sequence 2.13 might be converted into 2.14, as well as into 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.

```
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 [32].

Some examples of stanrard 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 (**f**, **g**) indicates that procedure **f** calls procedure **g**. By making the call graph harder interpret, obfuscators can hinder the reverse engineers capability of understanding the program behaviour [32]. To do so one could:

- **Inline functions**. The code belonging to a subfunction is merged into the code of its caller. If a subfunction 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 adding extra fake arguments, reordering arguments 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 will reside in a single region [32].

Violating this assumption introducing unconditional branches that break sequential locality and temporal locality of multiple basic blocks makes manual analysis more difficult. 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* (branch) and *CALL* (save instruction pointer and branch) 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 using dynamically computed branch addresses or by emulating them.

For example the instruction *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 As already seed when talking about anti-disassembler techniques, obfuscation can also exploit operating system primitives and structures. 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) [32].

Subroutine Reordering *Subroutine reordering* in 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 [31].

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*). We already briefly talked about this specific combination when talking about the *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 [32].

Simultaneous Control-Flow and Data-Flow Obfuscation

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

Inserting Junk Code This technique consists in introducing a dead code block (meaning that it will never be executed at runtime) between two other code blocks. Typically used in conjunction with *opaque predicates*, this technique is used to hinder 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 [32].

```
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

The listing 2.21 presents an example of this technique. More precisely the instruction at line 2 (*xor eax, eax*) zeroes the *EAX* register setting clearing the zero flag (it is set to 0); therefore the conditional jump (*jz 9*) at line 3 is always taken at runtime. The immediately next 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 to hide the true basic blocks 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 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 [32].

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 [31]. 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 code that is understood by a virtual machine. More specifically, the instruction set from the source machine is converted into a new instruction set, understood by the target virtual machine. This can be done using multiple types of virtual machines with different instruction sets. This means that a specific block of, for example, Intel x86 instructions can be converted into a different instruction set for each machine, preventing an analyst/attacker from recognizing any generated virtual opcode after the transformation from x86 instructions [32].

Usually, some specific blocks of the program's code are virtualized (and not the whole program) and inserted back into the program alongside the associated interpreter. At run time, the

interpreter assumes execution control and translates the virtualized code back to the original byte code.

When an analyst/attacker 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 which is not recognized by him or any other special decompiler. This will force the attacker to identify how each opcode is executed and how the specific virtual machine 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 it reassembles the code [31].

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.

In this approach, 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 [29].

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 [20].

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

Encryption of malware code is often used in conjunction with the use of packers.

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 [8].

As stated by Perdisci, et al [33], more than 80% of the new malware currently discovered are actually packed versions of already existing malware.

Packers are used to compress the original file into a smaller size and, moreover, encryption is sometimes applied to the compressed version of the file in order to make the unpacking process more difficult.

However, it is not uncommon to see malware authors writing and using custom packers. This fact can be used 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. On the other hand, many malware authors frequently use commercial, and readily available, packers to generate malware variants.

Oligomorphic Malwares

Malware authors tried to overcome the short comings of encrypted malware developing malware 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 different decryptors [20].

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 [29]. Still, signature based techniques are not an effective approach to detect *oligomorphic malware*.

The virus named *Whale* was the first known malware to make use of this technique. It carried a few dozens of different decryptors and picked one randomly at infection time [8].

Polymorphic Malwares

The *oligomorphic* malware limitations lead malware authors to develop a more advanced type of malware called *polymorphic*.

In *polymorphic* malware countless numbers (millions) of distinct decryptors can be generated by using obfuscation methods including, for example, *dead-code insertion*, *register reassignment*, *subroutine reordering*, *instruction substitution*, *code transposition/integration*, etc. to avoid signature based detection [29].

Polymorphic malware, like *oligomorphic* malware, consists of two parts: the malware encrypted main body and the decryptor. The decryptor is again run once 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. The malware appearance is thus changed at each infection [20].

In order to have a wide range of decryptors, *polymorphic* malware typically use a powerful toolkit called 'the *Mutation Engine* (*MtE*)'. In particular, during malware replication, the mutation engine is used to create the new decryptor which is appended to the new malware variant code. The mutation engine is, in fact, responsible for rearranging the decryptor code using different obfuscation techniques in order to prevent signature based detection.

Even though *polymorphic* malware 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, the tool 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 [8].

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* malware.

The first known malware to exhibit *polymorphism* is called *1260* and was written in 1990.

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 *polymorphic* malware, uses obfuscation techniques to create new variants of the original malware in order to evade detection [29].

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 [20].

However, *metamorphic* malware, in order to efficiently evolve its code it need 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 arbitrary increasing its code size is also challenging [8].

Moreover, *metamorphic* malware is also capable of interleaving its own code inside host programs, thus making detection even harder.

The first malware to exhibit metamorphic behaviour was called *Win95/Regswap* and was developed in 1998.

Chapter 3

Detection Techniques

Malware detection is the process of identifying malicious code from benign code. This is done in order to protect systems and being able to recover from the malicious code effects [8].

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, it doesn't mutate/change significantly at infection/replication time [20].

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 this type of evolving malware.

Commercial Portable Executable (PE) malware detectors consist of a hybrid of static and dynamic analysis engines. Static detection - which is fast and effective at detecting a large fraction of malware - is usually first employed to flag suspicious samples. Static detection involves analysing the raw PE image on disk and can be performed very quickly, but it is vulnerable to code obfuscation techniques.

Dynamic detection, by contrast, requires running the PE in an emulator and analysing behaviour at run time. When dynamic analysis works, it is less susceptible to code obfuscation, but takes substantially greater computational capacity and time to execute than static methods. Moreover, some files are difficult to execute in an emulated environment, but can still be statically analysed. Consequently, static detection methods are typically the most critical part of an endpoint's malware prevention pipeline.

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 [8].

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. Programs and files 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 [34].

Signature-based detection relies on *signatures*, represented by 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 families with respect to benign programs.

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. Anti-malware tools in turn scan the files in the target system and consider as malicious any file that matches one of the known signatures [8]. 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 *wildcards* characters to detect slight variations. Some second generation (evolving) malwares have been detected in the past by using wildcards, e.g. *W32/Regswap*.

This approach is fast, easy to use and has a high positive rate, however, since the number of known malwares is increasing so fast, it is quickly becoming time-consuming, expensive and impractical. Moreover, this is a completely reactive technique which is unable to counter threats/attacks from new malwares families/variants until they cause damages. Additionally, most second-generation malwares are able to escape this type of detection [20].

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 expressions 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 [35].

Yara rules can be generated either manually or automatically. Generating rules manually obviously requires high expertise, whereas generating them automatically using 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 [1].

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 [36].

Yara Rules syntax

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

```
rule RuleName
{
  meta:
  description = "description of rule"
  author = "name"
  date = "dd/mm/yyyy"
  reference = "url"

  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 keywork '*rule*', followed by the actual *RuleName*, which is the rule identifier. The *RuleNames* 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 [37].

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 since that is not their purpose [38].

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 [35].

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 (ASCI) 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_text_string = "text here"
  $my_hex_string = { E2 34 A1 C8 23 FB }

  condition:
  $my_text_string or $my_hex_string
```

```
}
```

Listing 3.2. YARA Hexadecimal

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

- ***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

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

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

- *Text Strings*: Text strings are generally readable sequences of ASCII characters which are then matched in the condition section [35].

Example of an ASCII-encoded, case-sensitive string:

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

  condition:
  $text_string
}
```

Listing 3.6. YARA Text Strings example

Additionally, to specify how *YARA* should search for strings, some modifiers can be added at the end of a string definition. Moreover, even more than one modifier can be used in combination to keep rule as simple and readable as possible. 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*), something which is typical 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

- *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

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

  The following rule will search for all the possible base64 permutations of the string "This program cannot":

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

  condition:
  $a
```

```
8   }
9
```

Listing 3.10. YARA Base64 encoded Strings

- *Regular Expressions*: Starting from version 2.0 YARA has 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 [1].

  Example:

```
1   rule RegExpExample
2   {
3     strings:
4     $re1 = /md5: [0-9a-fA-F]{32}/
5     $re2 = /state: (on|off)/
6
7     condition:
8     $re1 and $re2
9   }
10
```

Listing 3.11. YARA Regular Expression

Conditions section The last section of *YARA rules*, which is the only required one, contains the rule conditions that determine when the rule gets triggered. These conditions are Boolean expressions similar to those used in programming languages [38]. Through the use of all the usual logical and relational operators, conditions can be made arbitrary complex in order to accommodate for the specific author needs [35].

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 but with a # character in place of the initial $ character.

  For example:

```
1   rule CountExample
2   {
3     strings:
4     $a = "dummy1"
5     $b = "dummy2"
6
7     condition:
8     #a == 6 and #b > 10
9   }
10
```

Listing 3.12. YARA Count Example

- **Check String at specific offset/in offset range**: Sometimes we need 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 will find whether the '*a*' string is located at offset 100 and '*b*' at offset 200 of the running process.

```
1   rule AtExample
2   {
3     strings:
4     $a = "dummy1"
5     $b = "dummy2"
```

```
6
7     condition:
8     $a at 100 and $b at 200
9   }
10
```

Listing 3.13. YARA At Example

– The rule in listing 3.14 will find the '*a*' in the memory location between *0* and *100* and '*b*' between *100* and *filesize* of the running process.

```
1   rule InExample
2   {
3     strings:
4     $a = "dummy1"
5     $b = "dummy2"
6
7     condition:
8     $a in (0..100) and $b in (100..filesize)
9   }
10
```

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:

```
1   rule FileSizeExample
2   {
3     condition:
4     filesize > 200KB
5   }
6
```

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:

```
1   rule OfExample
2   {
3     strings:
4     $a = "dummy1"
5     $b = "dummy2"
6     $c = "dummy3"
7
8     condition:
9     2 of ($a, $b, $c)
10  }
11
```

Listing 3.16. YARA Of Example

Additional modules *YARA*'s code 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 [37]. 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

- **YARA with PEiD** *YARA* can also be integrated with *PEiD* to check what packer was used to compile the malicious/suspicious executable [37].

Yara Rules Advantages and Disadvantages

1. ***Advantages*** of *Yara rules*: *Yara rules* offer several advantages over other malware analysis techniques. Here are some of the most notable ones [1]:

   - *Yara rules* allow malware analysts to write flexible and custom rules in an easy and efficient way.
   - *Yara rules* are an open standard which work 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 generate *Yara rules* easily and efficiently.
   - There are various public repositories of *Yara rules* which offer readily available rules for malware analysis.

2. ***Limitations*** of *Yara rules*: *Yara rules*, however, have also some limitations. Here are some of the most notable ones [1]:

   - *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 malware which matches known malware signature. 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 [1].

The *yarGen* authors encourage its use as a starting point for rule construction, followed by manual adjustments and to refine *yarGen*'s output [36].

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, with 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 [1].

1. *YaraGenerator* tool *advantages* :
    - It can generate specialised rules for specific file formats.
    - It supports the use of PE (portable executable) modules, with 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 files.

- 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 for a while now.

***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 *Yara rules* produced contain a list of hexadecimal strings to be compared against suspicious files looking for similarities in their byte-sequences [1].

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 [36].

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 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 [36].

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.

Consider if to add something about "Manually generated Yara Rules + Open Source Yara Rules databases".

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 even detecting unknown malware variants [8].

3.4 Behavioural Based Detection

Behavioural-based malware detection is based on 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 identified as malware if their behavioural pattern is similar to the *behavioural signature* of a known malware [8].

Behavioural-based detection is for the most part immune to obfuscation attempt. However, being based on the time consuming dynamic analysis and on the challenging task of determining the unsafe activities and behaviours to consider within the environment, its applicability is limited.

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 [39].

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 [40].
- *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 [41].

Heuristic analysis is a promising technique for the detection of unknown malware, particularly for encrypted and polymorphic variants [20].

Nowadays *heuristic* analysis 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 [8].

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 on top of the use of manually generated signatures [44].

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 misclassification as measured by some loss criterion, which measures the deviation of predictions from their actual ground truth. 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, the learned detection function should work well on unseen samples [43].

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 [42]. There are various possible ways to statically craft feature vectors, for example:

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

The process of transforming a sample file to its numerical feature representation is called *feature extraction* and consists of some numerical transformation that preserves aggregate and fine-grained information throughout each sample [43].

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 [42].

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 [34].

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 [20].

In the following I will focus on the following recent static ML-based malware detection methods:

- 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**, [43], 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 thus making it difficult to include it as input features. Therefore, the authors proposed exploiting the metadata from threat intelligence feeds as auxiliary targets in the 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 [43] 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. The authors ultimately stated that this approach yielded a considerable improvement in performance on the main detection task.

Implementation Details The model presented in [43] 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 vector 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) [43].

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 malware/benign output of the network $\hat{y}^{(i)} = f_{mal}(\mathbf{x}^{(i)})$ and the malicious 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 [43].

Vendor Count Loss The authors investigated the use of the total number of 'malicious' reports for a given sample from the vendor aggregation service as an additional target; the rationale

being that a sample with a higher number of malicious vendor reports should be more likely to be malicious. In order to properly model this target, they require a suitable noise model for count data. A popular candidate is a Poisson noise model, parametrized by a single parameter μ, which assumes that counts follow a Poisson process, where μ is the mean and variance of the Poisson distribution. The probability of an observation of y counts conditional on μ is

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

In their problem they expected the mean number of positive results for a given sample to be related to the file itself, they attempted to learn to *estimate* μ conditional on each sample $\mathbf{x}^{(i)}$ in such a way that the likelihood of $y^{(i)}|\mu^{(i)}$ is maximized (or, equivalently, the negative log-likelihood is minimized). Denote the output of the neural network with which we are attempting to estimate the mean count of vendor positives for sample i as $f_{cnt}(\mathbf{x}^{(i)})$. Note that under a non-distributional loss, this would be denoted by $\hat{y}^{(i)}$, however since they are fitting a parameter of a distribution, and not the sample label y directly, they used different notation. By taking some appropriate activation function $a(\cdot)$ that map $f_{cnt}(\mathbf{x}^{(i)})$ to the non-negative real numbers, they can write $\mu^{(i)} = a(f_{cnt}(\mathbf{x}^{(i)}))$. The authors use an exponential activation for a, though one could equally well employ some other transformation with the correct output range, for instance the ReLU function.

Letting $y^{(i)}$ here denote the actual number of vendors that recognized sample $\mathbf{x}^{(i)}$ as 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 they refer to as the *Poisson* or *vendor count* loss. In practice, they ignore the $\log(y^{(i)}!)$ term when minimizing this loss since it does not depend on the parameters of the network.

A Poisson loss is more intuitive for dealing with count data than other common loss functions, even for count data not generated by a Poisson process.

The authors also implemented a Restricted Generalized Poisson distribution - a slightly more intricate noise model that accomodates dispersion in the variance of vendor counts.

$$P(y|\alpha, \mu) = \left(\frac{\mu}{1+\alpha\mu}\right)^y (1+\alpha y)^{y-1} \exp\left(\frac{-\mu(1+\alpha y)}{1+\alpha\mu}\right)/y!. \tag{3.4}$$

When $\alpha = 0$ this reduces to 3.2. $\alpha > 0$ accounts for over-dispersion, while $\alpha < 0$ accounts for under-dispersion. Note that in our case α, like μ, is estimated by the neural network and conditioned on the feature vector, allowing varying dispersion per-sample. The new loss function becomes:

$$\begin{aligned} L_{gp}(X, Y) = -\frac{1}{M}\sum_{i=1}^{M} [&y^{(i)}(\log\mu^{(i)} - \log(1+\alpha^{(i)}\mu^{(i)})) + \\ &(y^{(i)}-1)\log(1+\alpha^{(i)}y^{(i)}) \\ &- \frac{\mu^{(i)}(1+\alpha^{(i)}y^{(i)})}{1+\alpha^{(i)}\mu^{(i)}} + \log(y^{(i)}!)], \end{aligned} \tag{3.5}$$

where $\alpha^{(i)}$ and $\mu^{(i)}$ are obtained as transformed outputs of the neural network in a similar fashion as $\mu^{(i)}$ is obtained for the Poisson loss. In practice, as for the Poisson loss, the term related to $y!$ is dropped since it does not affect the optimization of the network parameters [43].

Per-Vendor Malware Loss The authors 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 a target in addition to the malware target by adding an extra fully connected layer per vendor followed by a sigmoid activation function to the end of the shared architecture. They employed a binary cross-entropy loss per vendor during training. This differs from the vendor count loss presented above in that each high-coverage vendor is used as an individual binary target, rather than being aggregated into a count. The aggregate *vendors* loss L_{vdr} for the $V = 9$ selected vendors is simply the sum of the individual vendor losses:

$$\begin{aligned} L_{vdr}(X, Y) &= \frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{V} l_{vdr}(f_{vdr_j}(\mathbf{x}^{(i)}), y_{v_j}^{(i)}) \\ &= -\frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{V} y_{v_j}^{(i)} \log(\hat{y}_{v_j}^{(i)}) + (1 - y_{v_j}^{(i)}) \log(1 - \hat{y}_{v_j}^{(i)}), \end{aligned} \tag{3.6}$$

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 $\mathbf{x}^{(i)}$ [43].

—

Malicious Tags Loss The authors exploit information contained in family detection names provided by different vendors in the form of malicious tags. They defined each tag as a high level description of the purpose of a given malicious sample. The tags used as auxiliary targets in their experiments are: *flooder*, *downloader*, *dropper*, *ransomware*, *crypto-miner*, *worm*, *adware*, *spyware*, *packed*, *file-infector* and *installer*.

They created these tags from a parse of individual vendor detection names, using a set of 10 vendors which provided high quality detection names. Once they have extracted the most common tokens, they filtered them to keep only tokens related to well-known malware family names or tokens that could easily be associated with one or more of these tags. The authors then created a mapping from tokens to tags based on prior knowledge.

Annotating the dataset with these tags, allowed the authors to define the tag prediction task as 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. They introduced this prediction task in order to have targets in their loss function that are not directly related to the number of vendors that recognize the sample as malicious.

In order to predict these tags, they used a *multi-headed* architecture in which they added two additional layers per tag to the end of the shared base architecture, a fully connected layer of size 512-to-256, followed by a sigmoid activation function. Each tag t_j out of the possible $T = 11$ tags has its own loss term computed with binary cross-entropy. Like the per-vendor malware loss, the aggregate tag loss is the sum of the individual tag losses [43]. For the dataset with M samples it becomes:

$$\begin{aligned} L_{tag}(X, Y) &= \frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{T} l_{tag}(f_{tag_j}(\mathbf{x}^{(i)}), y_{t_j}^{(i)}) \\ &= -\frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{T} y_{t_j}^{(i)} \log(\hat{y}_{t_j}^{(i)}) + (1 - y_{t_j}^{(i)}) \log(1 - \hat{y}_{t_j}^{(i)}), \end{aligned} \tag{3.7}$$

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 issued by the network for that value.

Dataset used The authors collected two datasets of PE files and associated metadata from a threat intelligence feed; a set for training/validation and a test set.

They extracted 1024-element feature vectors for all those files using feature type described in [] and derived an aggregate malicious/benign label using a 1-/5+ criterion as described above.

ROC curves. They also focused on improvements in detection at very low FPR of 0.1% or below, where we see the most dramatic improvements, since several publications by anti-virus vendors suggest that 0.1% or lower is indeed a practical FPR target for most deployment scenarios [43].

Final Loss + Experiment Evaluation Each model used a loss weight of 0.1 on the aggregate malicious/benign loss and 0.1 on each auxiliary loss, i.e. when we add K targets to the main loss, the final loss that gets backpropagated through the model becomes

$$L(X,Y) = L_{mal}(X,Y) + 0.1\sum_{k=1}^{K} L_k(X,Y). \tag{3.8}$$

3.6.2 Automatic Malware Description via Attribute Tagging and Similarity Embedding

(Abstract) Although powerful for conviction of malicious artefacts, machine learning detection methods do not produce any further information about the type of threat that has been detected neither allows for identifying relationships between malware samples. With this work the authors address the information gap between machine learning and signature-based detection methods by learning a representation space for malware samples in which files with similar malicious behaviours appear close to each other. They did so by introducing a deep learning based tagging model trained to generate human-interpretable semantic descriptions of malicious software, which, at the same time provides potentially more useful and flexible information than malware family names.

They show that the malware descriptions generated with the proposed approach correctly identify more than 95% of 11 possible tag descriptions for a given sample, at a deployable false positive rate of 1% per tag. Furthermore, they used the learned representation space to introduce a similarity index between malware files, and empirically demonstrated using dynamic traces from files' execution, that is not only more effective at identifying samples from the same families, but also 32 times smaller than those based on raw feature vectors [44].

—

(Introduction) Whenever one or more malicious files are found in a computer network, the first step towards remediation is to understand the nature of the attack in progress. Knowing the malicious capabilities associated with each suspicious file gives important context to network defenders which helps them define and prioritize counter-measures.

Generally, anti-virus (AV) or anti-malware solutions provide a *detection name* when they alert about potentially harmful files detected in a machine as a way to provide this context. These detection names usually come from specific signatures written by reverse engineers to identify particular threats, therefore encoding expert knowledge about a given malware sample. While this is theoretically useful for categorizing known malware variants, differing malware naming conventions among vendors have led to detection names that are inconsistent and highly vendor-specific. The problem of inconsistent naming conventions has been compounded due to more feature-rich malware and increased quantities of threats over time. Moreover, some detection names serve only as unique identifiers and do not provide actionable information about what type of harm the malicious sample could do if it infects a system.

When a novel malware variant appears, applying existing detection names, or even measuring similarity with known malicious files is problematic, since current rule-based signatures will likely not trigger on these variants at all. Machine learning (ML) malware detectors have the potential to identify these new malware samples as malicious, but generally do not provide further information about the type of threat encountered neither on how it relates with the universe of known malware.

The authors propose to use Semantic Malware Attribute Relevance Tagging (SMART) to approach malicious software description. In contradistinction to prior malware (family) detection

names, this semantic malware attribute tags approach yields human interpretable, high level descriptions of the capabilities of a given malware sample. SMART tags are related to malware family names in the sense that they attempt to describe how a piece of malicious software executed and the intent behind it. However, unlike malware family names, malware tags are non-exclusive, meaning that one malware campaign (or family) can be associated with multiple tags and a given tag can be associated with multiple malware families.

The number of tags is also inherently bounded by types of malicious behaviour and chosen granularity in description. Thus, a fixed number of tags can roughly describe all malicious samples, even when the number of malware families increases dramatically. Because of this, the tagging approach makes the task of malware description suitable to be addressed with machine learning methods.

The authors derived tags by leveraging the underlying knowledge encoded in detection names from different anti-malware vendors in the industry, although the general framework applies whenever multiple analyses of the same file are available.

Using their derived tags, the authors then trained a multi-label deep neural network to automatically predict tags for new (unseen) files in real time. Their approach only assumes access to the files' static binary representations. They found that their network yields impressive performance on the tag prediction task. It does so by learning a low dimensional Euclidean representation space in which malware samples with similar characteristics are close to each other. During the model training phase they used binary files and malware descriptions generated by multiple expert systems to train a machine learning model, that can later be deployed to automatically produce descriptive tags for new files in a fraction of a second.

The proposed approach of describing malicious capabilities by learning a low dimensional embedding space (and associated embedding function) for malicious files enables to compare malware samples by *semantic similarity* in terms of *type* of malicious content. This compressed representation further allows for efficient indexing, searching and querying of malware corpora along explainable dimensions. This ability is particularly useful for identifying new samples of a novel malware campaign from which we only have identified a small number of samples. This similarity metric in latent space also opens the door to novel applications in the context of endpoint detection and response (EDR) technologies, such as natural language queries, mapping how a given (potentially novel) malware campaign relates to and compares with known malware, and alerts prioritization based on malicious content, among others.

The authors introduced the task of automatic malicious tag prediction for malware description, and proposed a Joint Embedding neural network architecture and training methodology that allow to generate descriptive tags for malware files with high true positive rates at low false positive rates in a static fashion (without executing the software). Their neural network learnt a compact, yet expressive representation space for binary files which is informed by their malicious capabilities. They proposed a novel file-file similarity index based on the representation space of their neural network that enables for efficient indexing and searching in large malware corpora along interpretable dimensions.

—

Background and Related Work

Malware Family Categorization Identifying the family and variant of a particular malicious sample can provide important intelligence to the end user, administrator or security operator of a system about what type of attack might be underway. This extra contextual information can help define a remediation procedure, identify possible root causes, and evaluate the severity and potential consequences of the attack. In fact, numerous vendors provide in their websites detailed information about popular family/variant information, with associated description of what that variant does, and suggest how a particular piece of malware can be removed. Without such identifying information, we are left only with the offending file itself as its own description. Unlike some reverse engineering effort is taken it is difficult to discern much about the internals of the file.

The idea to identify all malware under a consistent family naming scheme across multiple vendors has been around for decades.

The quantity of new malware samples that security vendors' labs receive has increased dramatically, to millions per month. Some of these samples are variations of previously known malware, while others take code from older campaigns and re-purpose it for new tasks. Yet still others, are entirely novel types of malware. In this scenario it becomes practically unfeasible to manually and consistently group each malicious file into a well defined hierarchy of families. Even the arguably simpler task of assigning a malware file into an existing family has also become much harder, as malware became more resistant to signature-focused detection, thanks to advanced obfuscation measures such as polymorphism and metamorphism, packing and obfuscation, recompilation, and self-updating.

The increasing quantities of malware samples and the resilience to signature methods caused the security community to start using more flexible analytic tools than signatures designed only to identify a single malware variant, and increasingly rely on dynamic analysis and more generic signatures when possible. While generic signatures offer an advantage for malware conviction, the nature of this approach makes it more difficult to organize malware names into families. Moreover, the quantities of malicious files to be analysed has led to less structured categorizations and greater inconsistencies between vendors. In particular, modern security vendor detection names typically fall into one of four categories, containing varying amounts of information about the threat family to which a malware sample belongs.

- *Traditional family based*: Names are associated with unique and distinctive attributes of the malware and its variants. Malware classified under these names usually have a larger amount of original source code or a novel exploit mechanism and often come from the same origins. This not only fives them a distinctive attribute that can aid in the classification, but also requires researchers to put forth more effort to analyse their inner-workings. These types of detection names are often of high-quality and have more consistency across vendors.

  Today, these detection names are most often seen in parasitic file-infectors and specific botnet campaigns with distinctive attributes, e.g. *Virut*, *Sality*, *Conficker*, etc. These types of names usually use a suffix to identify specific variants, which often denotes a revision to the malware or change in the configuration data for use in a different campaign.

- *Technique based*: These types of detection names group together malware that may come from different origins and/or have multiple authors but share a common method or technique. For example, many executable *autorun* worms have been written in the past using languages like Visual Basic 6 that change explorer's file and folder view settings to hide filename extensions and employ an icon resource similar to that of a popular document format. Due to the relatively low complexity of the infection method many amateurs copied this technique resulting in a large amount of similar malware that was not necessarily of the same origin, neither tries to perform the same action in the host machine.

  Some anti-malware solutions would generically detect and classify many of these malware samples under the same generic family name, where other vendors may have defined different more specific criteria for each family classification based on other attributes of the payload. When a generic family name provided by the AV vendor, they often times replace the detection name suffix with a partial hash of the file data in order to identify a specific sample. The difference in detection methods employed often results in less consistency in detection names across vendors.

- *Method based*: This type of detection name simply denotes the detection technology used to detect the malware sample. Some detection names can simply be that of a patented technology, project, or internal code name specific to the AV vendor, indicating the use of heuristics, ML, or real-time detection technologies like cloud look-ups. In these cases the detection name is not that of a malware family, but that of the method that was utilized to detect the sample.

- *Kit based*: AV vendors will often use more generic family names for detecting malware that has been generated by a known kit. These kits are often referred to as grey hat tools, as

they can be used both offensively by penetration testing teams and by malware authors. Many of these kits obfuscate their payloads in an attempt to circumvent detection by AV software. Detection names in this category tend to not describe the origins or functionality of the specific malicious payload, but instead identify methods used by the kit or tool to obfuscate or hide their payload.

In [], Marcos et al. proposed AVClass: a malware labelling tool that uses data-mining techniques to distill family names for malware samples by combining detection names from multiple anti-malware solutions. Complementary to AVClass, Perdisci ey. al. proposed an automated technique in [] which relies on individual detection names from multiple vendors, for evaluating the quality of a given malware clustering. The authors followed the idea of combining detection names from multiple vendors to better understand the nature of malware samples, but instead of trying to fit each new sample to a naming scheme with mutually exclusive hierarchical categories such as families, they proposed an alternate approach to describing the functionality and the relationship between malicious samples by using attribute tags. A set of attribute tags describes a piece of malware though easy to interpret properties, and can be thought of as a soft-family classification, since it describes the sample and relates it with other samples described with the same (or an overlapping) set of tags. The advantage of the tagging approach is that it does not presume a partition on the malware space by genealogy, while providing potentially more actionable information about a malware sample.

Semantic Attribute Tagging Semantic attribute tagging refers to the association of samples with key-words that convey various types of high-level information about their content. These tags can later be used to interpret or summarize the content of the sample, for information retrieval in a large database of samples or for clustering, among others.

Semantic attribute tagging has two important characteristics worth considering:

1. it can convey a lot of identifying information about a sample, even if the sample is novel.
2. semantic tags can be stored, structured, and retrieved in a human interpretable manner.

Both of these characteristics are appealing in a commercial computer security use case where the type of the threat can be roughly identified by a description that makes sense to security researchers and end users.

Multi-Label Classification Semantic attribute tagging relies on multi-label classification, wherein the authors aim to predict multiple labels simultaneously. There are several ways to do this, the most trivial of which is to learn one classifier per label. This naive approach is not efficient in the sense that one classifier does not benefit from what the other classifiers have learned about a given sample. Furthermore, it can be unfeasible from a deployment perspective, particularly as the number of labels grow. For correlated labels, a popular approach is to use a single classifier with multiple outputs, on per output label. The total loss for the classifier is obtained by adding together the loss terms across the model's outputs during training and optimizing over a multi-objective loss. Not only does this yield a more compact representation but it also improves classification performance over using independent classifiers. The authors used as baseline architecture a multi-label deep neural network architecture which exploits a shared representation of the input samples, and has multiple binary cross entropy loss functions atop stacks of hidden layers, or heads, with final sigmoid outputs - one per tag.

An alternative approach to multi-label classification, first introduced in the image tagging and retrieval literature, is to learn a compact shared vector space representation to which 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. At query time, a similarity comparison between vectors in this learned latent space is performed, e.g. via inner product, to determine likely labels. A variety of models could be employed to form a joint embedding, but crucially, the embedding is optimized across input modalities/labels. The authors used a joint embedding model that maps malware tags and executable files into the same low dimensional Euclidean space for the malware description problem.

Malware Analysis with Neural Networks In contrast to the authors' work which focuses on malware description and representation, most modern applications of deep learning have focused on malware detection. Saxe et al. [] applied deep neural network detection to feature vectors derived from 2-dimensional histogram statistics of PE files along with hashed delimited strings and hashed elements from the file header, including metadata and import tables. Further applications of deep learning exploiting similar feature sets have been used to categorize web content, office documents, and archive formats. Other types of features and classifiers have also been used for the task of PE malware detection. For instance, Raff et al. demonstrated in [] a way to effectively identify malware using solely an embedding of the first 300 bytes from the PE header. Later, Raff et al. proposed an embedding strategy which takes in the entire PE file for the same problem. In [], Bugra and Erdogan use a disassembler to retrieve the opcodes of the executable files and then a shallow network based on *WORD2VEC* [] to embed them into a continuous vector space. Afterwards, they train a gradient search algorithm based on Gradient Boosting Machines for the malware classification task.

Semantic Malware Attribute Tags The authors defined SMART tags (which they also refer to as *malicious* or *malware* tags) as potentially informative, high-level attributes of malicious or potentially unwanted software. These tags are loosely related to malware families, in the sense that they attempt to describe how a piece of malicious software executes and the intent behind it, but they do so in a more general and flexible way. One malware campaign (or family) can be associated with more than one tag, and a given tag is associated with multiple families. The authors defined a set of malicious tags $\mathcal{T}$, with $|\mathcal{T}| = 11$ different tags (or descriptive dimensions) of interest that they can use to describe malicious PE files: *adware*, *crypto-miner*, *downloader*, *dropper*, *file-infector*, *flooder*, *installer*, *packed*, *ransomware*, *spyware* and *worm*. They chose that particular set of tags so that they can generate concise descriptions for most common malware currently found in the wild.

Since malware tags are defined at a different level of abstraction than malware families, the authors can bypass the problem of not having a common naming strategy for malicious software, and thus exploit the knowledge contained in multiple genealogies fenerated from different sources in a quasi-independent manner: detection technologies, methodologies, etc. It becomes irrelevant if one source identifies a sample as being part of the *Qakbot* family while another calls it *Banking Trojan* so long as there is a way to associate those two correctly with the *spyware* tag. Furthermore, this approach allows to exploit the fact that some sources might have stronger detection rules for certain kinds of malware.

Tag Distillation from Detection Names High quality tags for malware samples at the scale required to train deep learning models can be prohibitively expensive to create manually. Instead, the authors relied on semi-automatic strategies that are noisier than manual labelling, but allow to label millions of files that can then be used to train a classifier. The authors proposed, for this purpose, a labelling function that annotates PE files using the previously defined set of tags by combining information contained in detection names from multiple vendors. They used family names from ten anti-malware solutions that are known to produce high quality detection names, as starting point. Using multiple anti-malware vendors is one possible strategy for labelling that leverages expert knowledge about a given sample. Nevertheless, the overall framework the authors propose can easily accommodate multiple other sources for labelling information, including manual and sandbox-derived (dynamic) analysis.

The labelling process consists of three main stages:

1. *token extraction*
2. *token-to-tag mapping*
3. *token relationship mining*

The token extraction phase consists in normalizing and parsing the multiple detection names and converting them in sets of sub-strings. In a similar way that the AVClass labelling tool does,

the token-to-tag mapping stage uses rules that associate relevant tokens with the set of tags of interest. These association rules were created from expert knowledge by a group of malware analysts. Finally, they extend this mapping by mining statistical relationships between tokens to improve tagging stability and coverage.

Tags Prediction - Implementation Details The goal of the authors' work is to flexibly predict important malware qualities from static features, which they more formally define as multi-label classification problem, since zero or more tags from the set of T possible tags $\mathcal{T} = \{t_1, t_2, ..., t_T\}$ can be present at the same time for a given sample. In order to predict these tags, they proposed two different neural network architectures, represented in **??**, which they referred to as *Multi-Head* (top) and *Joint Embedding* (bottom).

The *Multi-Head* architecture can be thought as an extension of the network used in [] to multiple outputs. It consists of a base topology that is common to the prediction of all tags, and one output (or "head") per tag. The base topology can be thought of a feature extraction (or embedding) network that transforms the input features $\mathbf{x}$ into low dimensional hidden vector $\mathbf{h}$, while each head is a binary classifier that predicts the presence or absence of each tag from it. Both parts of the architecture consist of multiple blocks composed of dropout, a dense layer, batch normalization, and an exponential linear unit (ELU) activation function. The only exceptions are the input layer, which does not use dropout, and the very last layer of each head, which uses a sigmoid activation unit to compute the predicted probability for each label.

The *Joint Embedding* model, as shown at the bottom of **??**, is introduced with three main purposes:

1. in an attempt to better model semantic similarities between tags;
2. to explicitly define a low dimensional space that allows to naturally measure similarities between files;
3. to have a more flexible architecture that can scale to larger number of tags.

This model maps both the labels (malware tags) and the binary file features $\mathbf{x}$ to vectors in a joint Euclidean latent-space. These embedding functions of files and tags are learnt via stochastic gradient descent in a way such 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. This architecture consists on a PE *embedding* network, a *tag embedding* matrix $\mathbf{E}$, and a *prediction layer*.

The PE embedding network learns a nonlinear 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,

$$\phi_\theta(\mathbf{x}) : \mathbb{R}^d \to \mathbb{R}^D. \tag{3.9}$$

The tag embedding matrix $\mathbf{E} \in \mathbb{R}^{T \times D}$ 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.

$$\phi_E(t) : \{t_1, ..., t_T\} \to \mathbb{R}^D. \tag{3.10}$$

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 the tag and the sample embeddings ($\mathbf{e}$ and $\mathbf{h}$ respectively) and produces a similarity score that is 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, the similarity score is the dot product between the embedding vectors. The output of the network $f_n(\mathbf{x}|\theta, \mathbf{E})$ then becomes,

$$\begin{aligned} \hat{y}_n = f_n(\mathbf{x}|\theta, \mathbf{E}) &= \sigma(\langle \phi_E(n), \phi_\theta(\mathbf{x}) \rangle) \\ &= \sigma(\langle \mathbf{E}_n, \mathbf{h} \rangle), \end{aligned} \tag{3.11}$$

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 $\mathbf{x}$.

The authors further constrained the embedding vectors for the tags such that:

$$||\mathbf{E}_n||_2 \leq C, n = 1, ..., T, \tag{3.12}$$

which acts as a regularizer for the model. They observed in practice that this normalization indeed leads to better results on the validation set. Unless stated differently they fixed the value of C to 1.

The authors also experimented using cosine similarity instead of a dot product as a similarity score between tags' and files' embeddings. However, they observed deteriorated performance on the validation set.

The authors' goal, for a given PE file, is to learn a distributed, low dimensional representation of it, that is both "close" to the embedding of the tags that describe it and to other PE files with similar characteristics. 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 becomes:

$$\begin{aligned}\mathcal{L} &= -\frac{1}{M}\sum_{i=1}^{M}\sum_{n=1}^{T} f_n(\mathbf{x}^{(i)}|\theta, \mathbf{E}) \log(t_n^{(i)}) + (1 - f_n(\mathbf{x}^{(i)}|\theta, \mathbf{E})) \log(1 - t_n^{(i)}) \\ &= -\frac{1}{M}\sum_{i=1}^{M}\sum_{n=1}^{T} \hat{y}_n^{(i)} \log(t_n^{(i)}) + (1 - \hat{y}_n^{(i)}) \log(1 - t_n^{(i)})\end{aligned} \tag{3.13}$$

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.

In practice, to get a vector of tag similarities for a given sample $\mathbf{x}$ with PE embedding vector $\mathbf{h}$ we multiply the matrix of tag embeddings $\mathbf{E} \in \mathbb{R}^{T \times D}$ by $\mathbf{h} \in \mathbb{R}^D$ and scale the output 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.

The authors noted that, because this model maps every tag into the same low dimensional space, it can explicitly account for similarities on the labels. That is, if two tags are likely to occur together, they will be mapped close to each other in embedding space.

Finally, if one compares both models in **??**, they are mathematically similar, with the main differences being the explicit modelling of the tag embedding step (and its regularization), the subsequent explicit modelling of the joint embedding space that allows to naturally perform label-to-label and sample-to-sample similarity searches, and the ability to use various distance functions in the prediction layer.

Evaluation of Tagging Algorithms There are different ways to evaluate the performance of tagging algorithms. Particularly, the evaluation can be done in a *per-tag* or a *per-sample* dimension. The former seeks to quantify how well the tagging algorithm performs on identifying each tag, while the latter focuses on the quality of the predictions for each sample instead.

In the *per-tag* case, one suitable way to evaluate the performance of the model is to measure 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). Also, since the target value for the n-th tag of a given sample is a binary True/False value ($t_n \in \{0, 1\}$), binary classification evaluation metrics such as 'Accuracy', 'Precision', 'Recall', and 'F-score' also apply. To compute these metrics, the output probability prediction needs to be binarized. For the binarization of the predictions, the authors choose a threshold independently for each tag such that the FPR in the validation set is 0.01 and use the resulting 0/1 predictions. The fact that the labelling methodology introduces label noise -

mostly associated with negative labels - makes *recall* the most adequate of these last four metrics to evaluate the tagging algorithm, since it ignores incorrect negative labels.

The *per-sample* evaluation dimension seeks to evaluate the performance of a tagging algorithm for a given sample, across all tags. Let $T^{(i)}$ be the set of tags associated with sample i and $\hat{T}^{(i)}$ the set of tags predicted for the same sample after binarizing the predictions. We can use the Jaccard similarity (or index) $J(T^{(i)}, \hat{T^{(i)}})$ as a figure of how similar both sets are. Furthermore, let $\mathbf{y} \in \{0,1\}^T$ be the binary target vector for a PE file, where $\mathbf{y}_n$ indicates whether the n-th tag applies to the file and $\hat{\mathbf{y}}$ be the binarized prediction vector from a given tagging model. The per-sample accuracy is then defined as the percentage of samples for which the target vector is equal to the prediction vector, i.e. all tags correctly predicted, or, in other words, the Hamming distance between the two vectors is zero. For an evaluation dataset with M samples one can use,

$$\text{Mean Jaccard similarity} = \frac{1}{M}\sum_{i=1}^{M} J(T^{(i)}, \hat{T}^{(i)}) = \frac{1}{M}\sum_{i=1}^{M} \frac{T^{(i)} \cap \hat{T}^{(i)}}{T^{(i)} \cup \hat{T}^{(i)}} \tag{3.14}$$

$$\text{Mean per-sample accuracy} = \frac{1}{M}\sum_{i=1}^{M} \mathbb{I}(\mathbf{y}^{(i)} = \hat{\mathbf{y}}^{(i)}) \tag{3.15}$$

as per-sample performance metrics for the tagging problem, where $\mathbb{I}$ is the indicator function which is 1 if the condition in the argument is true, and 0 otherwise.

Dataset and Training details The authors used 2 different dataset for training purposes: a medium size training set containing 10 *million* unique binary Windows Portable Executable (PE) files ($D_{train-M}$), and a large training set containing 76,204,855 unique PE files ($D_{train-XL}$). The smaller training set was used for experimenting with different architectures and validation purposes, while the larger one was used to train the final models, whose architectures performed best in the validation set when trained on the smaller dataset.

The authors trained the 2 model architectures previously described on the training dataset $D_{train-M}$ with different number of layers and number of nodes per layer, each for 200 epochs using Adam optimizer on mini-batches of 4096 samples and learning rate of $5 \cdot 10^{-4}$. The combination of those values that performed best were then used to train the model on $D_{train-XL}$.

The shared base topology of the final Multi-Head architecture consisted of an input feed-forward layer of output size 2048, followed by a batch normalization layer, an ELU nonlinearity and three blocks, each composed by dropout, a linear layer, batch normalization and ELU of output sizes 512, 128, and 32 respectively. Each output head is simply a linear layer (the same for each head) composed of the same type of basic blocks as are 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. Binary cross-entropy loss is computed as the output of each head and then added together to form the final loss.

The Joint Embedding architecture used the same base topology as the Multi-Head model for the embedding of the PE files into a 32 dimensional joint latent space. An embedding matrix ($\mathbf{E}$) of learnable parameters with size $T \times 32$ is used for the embedding of the tags. The authors used dot product to compute the similarity between the PE file embedding and the tag embedding followed by a sigmoid non-linearity to produce an output probability score. As before, the sum of the per-tag binary cross-entropy losses is used as the mini-batch loss during model training.

Tagging results

Per-Tag Results After training the two proposed architectures the authors proceeded to evaluate their performance on the test set D_{test}. They compared the per-tag true positive rate (TPR or recall) of both the Multi-Head and Joint Embedding architectures at a per-tag false positive rate (FPR) of 1%.

For computing both recall and F-score they binarized the output using a threshold such that the FPR in the test set is 1% for each tag.

Per-Sample Results Another way of analysing the results is to measure the percentage of samples for which the models accurately predicted all tags. The authors were also interested in knowing how many tags on average (out of the 11 possible tags) each model correctly predicted per sample. For this they measured both the Jaccard similarity and the per-sample accuracy of the predictions according to equations **??** and **??** respectively.

Latent Space (Joint Embedding) Analysis The Joint Embedding network supposedly learns a more informative internal representation of the binary files, guided by the ability to model, and therefore exploit tag relationships (labels structure) in the latent space.

This improved internal representation has two main advantages compared with the original feature representation:

1. it is highly compressed (32 dimensions versus 1024) which allows for indexing malware databases in a more efficient way, and perform similarity computations and nearest neighbour searches faster;

2. this representation also provides a more informative space where to measure files' similarities in terms of malicious capabilities that is aligned with the original underlying family structure of malware.

Malware-Tag Joint Embedding Space In order to understand the latent space learned by the Joint Embedding model the authors used t-SNE [] to reduce the dimensionality of the 32-dimensional latent space to a 2-dimensional representation.

As seen by the t-SNE output, the embeddings of the PE files labelled with the same tag tend to cluster together. Furthermore, the embeddings of the tag labels lie close to the clusters of samples they describe. This suggests that the Joint Embedding model has effectively learned to group file representations close to their corresponding tag representations, as intended.

It is also possible to perform file similarity searches: using one malware sample to retrieve other samples with similar malicious characteristics by simply retrieving files for which the embedding representation is close to the representation of the query sample. Furthermore, it can also be used for obtaining examples of malicious programs that fit a given tag description, where with given a combination of descriptive tags we can define a subspace within the embedding space containing those samples that closely match that particular tag combination of interest.

File-File Similarity Index The proposed representation of PE files in embedding space, along with the distance function defined in this space, allows to measure not only similarities between files' and tags' embeddings, but between files' representations as well.

Because of how the models were trained, two pieces of malware that are close to each other in latent space should exhibit similar malicious capabilities. This means that it is possible to use the latent representation of the files to measure how similar two pieces of malware are in terms of their capabilities. Moreover, the size of the file embedding representation (32 dimensions) is 32 times smaller than the original file feature representation (1024 dimensions). This makes the task of storing, indexing and querying large databases of malware more efficient.

Because the models are being indirectly trained with malware families' information generated by a group of (quasi-)independent "experts" (anti-malware vendors), via tagging decomposition, then the authors expected to be able to differentiate malware campaigns from each other, even if they are described by the same set of tags.

The authors identified the top $f = 13$ most prominent malware campaigns in the test set that can be convicted with high confidence. They noted that there are a number of families with the same tag description, for instance *remcos* and *emotet* families are both described with the *spyware* tag.

Next, the authors created the joint embedding representation for those files using the already trained Joint Embedding network.

Finally, they set up the task of f-way family classification via nearest neighbour search. It consisted in randomly sampling k files per family as anchor samples, and q files per family as query samples. Each of the $(f \cdot q)$ query samples was then predicted to belong to the same class as its closest anchor sample in feature space. A more descriptive representational space should separate malware families, i.e. pull those samples that belong to the same family close to each other while pushing away samples from other families, thus performing better on the nearest neighbour classification task.

The authors considered the neural network embeddings and the original 1024 features as feature spaces for this task. They randomly chose 23 query samples per family and varied the number of anchor samples k from 1 to 10. They repeated the sampling process (for both anchors and query samples), classification and evaluation 15 times per value of k to obtain uncertainty estimates for the accuracy results.

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

(Abstract) Machine learning (ML) classifiers used for static portable executable (PE) malware detection typically employ single numerical feature vector representations of each file as input with one or more target labels per file during training. However, there is much orthogonal information that can be gleaned from the *context* in which the file was seen.

The authors propose utilizing a static source of contextual information - namely the path of the PE file - as an auxiliary input to the classifier. While file paths are not malicious or benign in and of themselves, they provide valuable context for a malicious/benign determination.

Unlike dynamic contextual information, which requires high CPU and runtime overhead, file paths are available with little overhead and can seamlessly be integrated into a multi-view static ML detector, potentially yielding higher detection rates at very high throughput with minimal infrastructural changes.

The authors proposed a multi-view neural network, which takes feature vectors from the PE file content as well as corresponding file paths as inputs and outputs a detection score. They used a commercial-scale dataset of approximately 10 million samples - files and file paths from user endpoints of an actual security vendor network. They found that their model learned useful aspects of the file paths for classification [42].

—

(Introduction) Static detection methods have seen performance advancements recently, thanks to the adoption of machine learning, where highly expressive classifiers, e.g. deep neural networks, are fit on labelled data sets of millions of files. When these classifiers are trained, they use *feature vectors* - numerical descriptions of the static file content - as input but no auxiliary data. Dynamic analysis, however, works well precisely because of *auxiliary data* - e.g. network traffic, system calls, etc. - information that cannot be gleaned directly from the static content of the file.

The authors used file paths as orthogonal input information to augment static ML detectors. File paths are available statically, without any additional instrumentation of the OS, and are already used internally by malware analysts to correct and investigate mis-characterised detections.

Using file paths to augment detections on the surface seems potentially problematic, as file paths are not inherently malicious or benign. However, malware droppers often use file paths with certain characteristics for a variety of reasons. For example, a file path may be chosen to increase the likelihood that a user will execute a malicious PE masquerading as another application, to avoid disk scans, or to hide the files from a user's view. This results in a prevalence of certain types of directory hierarchies, and detectable naming characteristics, which can provide useful hints about the malicious/benign nature of a file, even when this is not immediately obvious from its content.

By including the file path as an auxiliary input, the authors are able to combine information about the file, via feature vectors, with information about how likely it is to see such a file in that specific location.

They focused their analysis on thee models:

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

The authors collected a commercial dataset of actual files and file paths scans on customer endpoints from a large anti-malware vendor, and used them to perform a time split validation ot their model.

Their multi-view classifier trained on both file content and the contextual file paths yielded statistically better results across the ROC curve and particularly in low false positive rate (FPR) regions [42].

—

Background and Related work (Static ML Malware Detection) The authors used in their work a concatenation of features derived from the content of a PE file as an input to a neural network, adding a second input which includes contextual information - namely the PE File path. The PE content input is passed though a series of hidden layers while the file path is passed through a convolutional embedding. Both inputs are ultimately concatenated together into a common 'stem' of hidden layers. The final malicious/benign output score is obtained by passing the final dense layer output (a 1-D scalar) through a sigmoid activation function [42].

—

(Learning from Multiple Sources) This approach utilizes *multiple* input types/modalities - one which describes the content of the malicious sample, in the form of a PE feature vector, and another which feeds the path of the file to an embedding which provides information on where that sample was seen. This technique is a type of *multi-view* learning. The majority of applications of multi-view learning are in computer vision, where the multiple views literally consist of views from different input cameras/sensors or different views from the same camera/sensor at different times [42].

—

Dataset used The authors collected training, validation and testing datasets from a prominent anti-malware vendor's telemetry. This telemetry contained the filepaths and SHA256 digests of portable executable (PE) files seen on their customer endpoints, along with time stamps and other metadata. The telemetry did not contain the raw files due to bandwidth and customer privacy considerations, and instead they used the SHA256 digests to look up and download available files from vendor aggregation services. Malicious/benign labels for these files were computed using a criterion similar to [45], but combined with additional propriety information to generate more accurate labelling. The filepaths were lower-cased for consistency. The data was split into training and test datasets based on the time samples were first seen in their telemetry [42].

—

Feature Engineering In order to use file paths in feed-forward neural network, the authors first needed to convert the variable length strings into numeric vectors of fixed length. They did that using a vectorization scheme, by creating a lookup table keyed on each character with a numeric value (between 0 and the character set size) representing each character. This transformation required their file paths to be trimmed to a fixed size, so they trimmed them to the last 100 characters.

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 [45]:

1. A 256-dimensional (16x16) 2D histogram of windowed entropy values per byte. A window size of 1024 was selected.
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 presented each sample as two feature vectors: A PE content feature vector of 1024 dimensions and a contextual file path feature vector of 100 dimensions [42].

—

Implementation details The model has two inputs, the 1024 element PE content feature vector, $\mathbf{x}_{PE}$, and the 100 element file path integer vector, $\mathbf{x}_{FP}$. Each distinct input is passed through a series of layers with their own parameters, θ_{PE} and θ_{FP}, for PE features and FP for filepath features respectively, and are jointly optimized during training. The outputs of the se layers are then joined (concatenated) and passed through a series of final hidden layers - a joint output path with parameters θ_O. The final output of the network consists of a dense layer followed by a sigmoid activation. The labelling convention uses 0 as a benign label and 1 as a malicious label, so sigmoid outputs close to 1 are more likely to be malicious than outputs close to 0, which are more likely to be benign. However, the threshold for malicious/benign determination can be set anywhere along the $(0.0, 1.0)$ range according to false positive rate (FPR) and detection rate (TPR) tradeoffs for the application at hand - a reasonable threshold is typically at or below 10^{-3} FPR.

The PE input arm θ_{PE} passes $\mathbf{x}_{PE}$ through a series of blocks consisting of four layers each: a Fully Connected layer, a Layer Normalization layer, a Dropout layer with a dropout probability of 0.05, and a Rectified Linear Unit (ReLU) activation. Five of these blocks are connected in sequence with dense layer sizes 1024, 768, 512, 512, and 512 nodes respectively in order.

The file path input arm θ_{FP}, passes $\mathbf{x}_{FP}$ - a vector of length 100 - into an Embedding layer that converts the integer vector into a $(100, 32)$ embedding. This embedding is then fed into 4 separate convolution blocks, that contain a 1D convolution layer with 128 filters, a Layer Normalization layer and a 1D sum layer to flatten the output to a vector. The 4 convolution blocks contain convolution layers with filters of size 2, 3, 4 and 5 respectively that process 2, 3, 4 and 5-grams of the input file path. The flattened outputs of these convolution blocks are then concatenated and serve as input to two dense blocks (same form as in the PE input arm).

The outputs from the fully connected blocks from the PE arm and the file path arm are then concatenated and passed into the joint output path, parametrized by θ_O. This path consists of dense connected blocks (same form as in the PE input arm) of layer sizes 512, 256 and 128. The $128D$ output of these blocks is then fed to a dense layer which projects the output to $1D$, followed by a sigmoid activation that provides the final output of the model.

The PE only model is just the P+ FP model but without the FP arm, taking input $\mathbf{x}_{PE}$ and fitting θ_{PE} and θ_O parameters. Similarly, the FP model is the PE+FP model but without the PE arm, taking input $\mathbf{x}_{FP}$ fitting θ_{FP} and θ_O parameters. The first layer of the output subnetwork is adjusted appropriately to match the output from the previous layer.

All models are fit using a binary cross entropy loss function. Given the output of the deep learning model $f(\mathbf{x}; \theta)$ for input $\mathbf{x}$ with label $y \in \{0,1\}$ and model parameters θ the loss is:

$$L(\mathbf{x}, y; \theta) = -y \log\left(f(\mathbf{x}; \theta)\right) + (1 - y) \log\left(1 - f(\mathbf{x}; \theta)\right). \tag{3.16}$$

Via an optimizer, the equation is solved for $\hat{\theta}$ the optimal set of parameters that minimize the combined loss over the dataset:

$$\hat{\theta} = \arg\max_{\theta} \sum_{i=1}^{M} L(\mathbf{x}^{(i)}, y_i; \theta), \tag{3.17}$$

where M is the number of samples in our dataset, and y_i and $\mathbf{x}^{(i)}$ are the label and the feature vector of the ith training sample respectively [42].

—

(Conclusion) Deep neural network malware detectors can benefit from contextual information from file paths, even when this information is not inherently malicious or benign. Adding file paths to the detection model did not require any additional endpoint instrumentation, and provided a statistically significant improvement in the overall ROC curve, throughout relevant FPR regions [42].

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 [20].

Chapter 4

Datasets

4.1 Ember Dataset

(Abstract) EMBER is a labelled benchmark dataset for training machine learning models to statically detect malicious Windows Portable Executable (PE) files. The dataset includes features extracted from 1.1M binary files: 900K training samples (300K malicious, 300K benign, 300K unlabelled) and 200K test samples (100K malicious, 100K benign). The authors also released open source code for extracting features from additional binaries so that additional sample features can be appended to the dataset. This dataset fills a void in the information security machine learning community: a benign/malicious dataset that is large, open and general enough to cover several interesting use cases [46].

—

(Introduction) Machine learning can be an attractive tool for either a primary detection capability or supplementary detection heuristics. Supervised learning models automatically exploit complex relationships between file attributes in training data that are discriminating between malicious and benign samples. Furthermore, properly regularized machine learning models generalize to new samples whose features and labels follow a similar distribution to the training data.

However, malware detection using machine learning has not received nearly the same attention in the open research community as other applications, where rich benchmark datasets exist. These include handwritten digit classification (e.g. MNIST), image labelling (e.g. CIFAR or ImageNet), traffic sign detection, speech recognition (e.g. TIMIT), sentiment analysis, and a host of other datasets suitable for training models to mimic human perception and cognition tasks. The challenges to releasing a benchmark dataset for malware detection are many, and may include the following: *Labelling challenges.* Unlike images, text and speech - which may be labelled relatively quickly, and in many cases by a non-expert - determining whether a binary file is malicious benign can be a time-consuming process for even the well-trained. The work of labelling may be automated via anti-malware scanners that codify much of this human expertise, but the results may be proprietary or otherwise protected.

The authors address this issue with the release of the Endgame Malware BEnchmark for Research (EMBER) dataset, extracted from a large corpus of Windows Portable Executable (PE) malicious and benign files. Samples are released together with the sha256 hash of the original file, and a label to denote whether the file is deemed to be malicious or benign [46].

4.1.1 PE File Format

The PE file format describes the predominant executable format for Microsoft Windows operating systems, and includes executables, dynamically-linked libraries (DLLs), and FON font files. The format is currently supported on Intel, AMD and variants of ARM instruction set architectures.

The file format is arranged with a number of standard headers (**??**), followed by one or more sections. Headers include the Common Object File Format (COFF) file header that contains important information such as the type of machine for which the file is intended, the nature of the file (DLL, EXE, OBJ), the number of sections, the number of symbols, etc. The optional header identifies the linker version, the size of the code, the size of initialized and uninitialized data, the address of the entry point, etc. Data directories within the optional header provide pointers to the sections that follow it. This includes tables for exports, imports, resources, exceptions, debug information, certificate information, and relocation tables. AS such, it provides a useful summary of the contents of an executable. Finally, the section table outlines the name, offset and size of each section in the PE file.

PE sections contain code and initialized data that the Windows loader is to map into executable or readable/write-able memory pages, respectively, as well as imports, exports and resources defined by the file. Each section contains a header that specifies the size and address. An import address table instructs the loader which functions to statically import. A resources section may contain resources such as required for user interfaces: cursors, fonts, bitmaps, icons, menus, etc. A basic PE file would normally contain a *.text* code section and one or more data sections (*.data*, *.text* or *.bss*). Relocation tables are typically stored in a *.reloc* section, used by the Windows loader to reassign a base address from the executable's preferred base. A *.tls* section contains special thread local storage (TLS) structure for storing thread-specific local variables, which has been exploited to redirect the entry point of an executable to first check if a debugger or other analysis tool are being run. Section names are arbitrary from the perspective of the Windows loader, but specific names have been adopted by precedent and are overwhelmingly common. Packers may create new sections, for example, the UPX packer creates *UPX1* to house packed data and an empty section *UPX0* that reserves an address range for runtime unpacking [46].

4.1.2 Static PE Malware Detection

(to be moved) Static malware detection attempts to classify samples as malicious or benign without executing them, in contrast to dynamic malware detection which detects malware based on its runtime behaviour including time-dependent sequences of system calls for analysis. Although static detection is well-known to be undecidable in general, it is an important protection layer in a security suite because when successful, it allows malicious files to be detected prior to execution.

Machine learning-based static PE malware detectors have been used since at least 2001, and owing largely to the structured file format and backwards-compatibility requirements, many concepts remain surprisingly similar in subsequent works. Schultz et al. [] assembled a dataset and generated labels by running through a McAfee virus scanner. PE files were represented by features that included imported functions, strings and byte sequences. Various machine learning models were trained and validated on a holdout set. Models included rules induced from RIPPER [], naive Bayes and an ensemble classifier. Kolter et al. [] extended this approach by including byte-level N-grams, and employed techniques from natural language processing, including *tf-idf* weighting of strings. Shafiq et al. [] proposed using just seven features from the PE header, motivated by the fact that most malware samples in their study typically exhibited those elements. Saxe and Berlin leveraged novel two dimensional byte entropy histograms that is fed into a multi-layer neural network for classification [].

Recent advances in end-to-end deep learning have dramatically improved the state of the art especially in object classification, machine translation and speech recognition. In many of these approaches, raw images, text or speech waveforms are used as input to a machine learning model which infers the most useful feature representation for the task at hand. However, despite successes in other domains, hand-crafted features apparently still represent the state of the art for malware detection in published literature. The state of the art may change to end-to-end deep learning in the ensuing months or years, but hand-crafted features derived from parsing the PE file may continue to be relevant indefinitely because of the structured format [46].

4.1.3 Malicious and benign datasets

(to be moved) PE-Miner aimed to produce a machine-learning based malware detector with high true positive rate (TPR) at low false positive rates (FPRs), with a runtime comparable to signature-based scanners of the day []. It was trained on a dataset of $\sim 1K$ benign files on the operating system, $\sim 10K$ malicious PE files from VXHeaven and $\sim 5K$ malicious PE files from Malfease. Unfortunately, many of the features were not disclosed publicly, some being deemed sensitive and protected under NDA. Several model types were evaluated on the dataset, and of those, it was discovered that the J48 decision tree algorithm provided the best performance. Notably, although many papers cite this work as one of the first performant non-signature-based methods, the lack of public dataset has resulted no real comparative study.

Shortly following, the Adobe Malware Classifier aimed to produce a malware classifier from only seven features: debug size, image version, relative virtual address of the import address table, export size, resource size, virtual size of the second section, and the total number of sections. A decision tree algorithm was trained and the resulting classifier released as a freely available tool. It has been suggested, however, that since the benign dataset largely comprised of Windows binaries, the resulting models is strongly biased towards a non-Windows vs. Windows rather than a malicious vs. benign problem. Moreover, unfortunately, the dataset consisting of $\sim 100K$ malicious and $\sim 16K$ benign files was never released for comparative research.

In contrast, the Microsoft Malware Classification Challenge 500MB dataset consisted of disassembly and byte-code of $\sim 20K$ malicious samples from 9 families. The largest family consisted of features from $3K$ samples, while the smallest family included only 42 samples. Since the conclusion of the competition, more than 50 research papers and theses cited the dataset. Unfortunately, the disassembly features are specific to IDA Pro disassembler, and the dataset contains no benign files.

Malware sharing services like VXHeaven provide an ample supply of malicious binaries. VirusTotal can be mined for supposed benign files using heuristics about the number of detections of vendor participants. However, large-scale file access rates in VirusTotal require a payed subscription. Regardless, an agreed-upon set of malicious and benign files for machine learning benchmark purposes is so far non-existent [46].

4.1.4 Data Description

In crafting the EMBER dataset, the authors considered several practical use cases and research studies, including the following [46].

- Compare machine learning models for malware detection.
- Quantify model degradation and concept drift over time.
- Research interpretable machine learning.
- Compare features for malware classification, particularly novel features not represented in the EMBER dataset. This requires an extensible dataset.
- Compare to featureless end-to-end deep learning. This may require code to extract features from a new dataset, or sha256 hashes to build a raw binary dataset to match EMBER.
- Research adversarial attacks against machine learning malware, and subsequent defence strategies.
- Leverage unlabelled samples via unsupervised learning for PE file representation or semi-supervised learning for classification.

4.1.5 Data Layout

The EMBER dataset consists of a collection of JSON lines files, where each line contains a single JSON object. Each object includes the following types in data:

- the sha256 hash of the original file as a unique identifier;
- coarse time information (month resolution) that establishes an estimate of when the file was first seen;
- a 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.

For convenience the dataset is comprised of raw features that are human readable. The code that produces a numeric feature vector required for model building from raw features is also provided.

The authors temporally split the training/test sets to mimic generational dependencies of both malicious and benign software. The coarse time stamps for one year of malicious and benign files may also allow for simple longitudinal studies. Including the sha256 hash of the original file allows researchers to link features to the raw binaries, including other metadata that may be available through file sharing sites like VirusShare or VirusTotal [46].

4.1.6 Feature Set Description

The EMBER dataset consists of 8 groups of raw features that include both parsed features and format-agnostic histograms and counts of strings. The authors made a distinction between *raw features* (the dataset provided) and *model features* (or *vectorized features*) derived from the dataset. *Model features* represent a feature matrix of fixed size used for training a model, representing a numerical summary of the raw features, wherein strings, imported names, exported names, etc., are captured using the feature hashing trick []. The feature matrix is not explicitly provided in the published dataset, but code is provided to convert raw features to model features to train a model [46].

Parsed features

The dataset includes 5 groups of features that are extracted after parsing the PE file. The authors leveraged the *Library to Instrument Executable Formats* [] as a convenient PE parser. *LIEF* names are used for strings that represent symbolic objects, such as characteristics and properties [46].

General file information The set of features in 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, or a signature, and the number of symbols [46].

Header information From the *COFF* header, the authors reported the timestamp in the header, the target machine (string) and a list of image characteristics (list of strings). From the optional header, the authors provide the target subsystem (string), DLL characteristics (a list of strings), the file magic 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. TO create model features, string descriptors such as DLL characteristics, target machine, subsystem, etc. are summarized using the feature hashing trick prior to training a model, with 10 bins allotted for each noisy indicator vector [46].

Imported functions The authors parse the import address table and report the imported functions by library. To create model features for the baseline model, the authors simply collect the set of unique libraries and use the hashing trick to sketch the set (256 bins). Similarly, they use the hashing trick (1024 bins) to capture individual functions, by representing each as a string such as *library:FunctionName* pair (e.g. *kernel32.dll:CreateFileMappingA*) [46].

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 [46].

Section information Properties of each section are provided and include the name, size, entropy, virtual size, and a list of strings representing section characteristics. The entry point is specified by name. To convert to model features, the authors used the hashing trick on (section name, value) pairs to create vectors containing section size, section entropy, and virtual size (50 bins each). They also used the hashing trick to capture the characteristics (list of strings) for the entry point [46].

Format-agnostic features

The EMBER dataset also includes three groups of features that are format agnostic, in that they do not require parsing of the PE file for extraction: a raw byte histogram, byte entropy histogram based on work previously published in [], and string extraction [46].

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 [46].

Byte-entropy histogram The byte entropy histogram approximates the joint distribution $p(H, X)$ of entropy H and byte value X. This is done as described in [], 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 their implementation, the authors used a window size of 2048 and a step size of 1024 bytes, with 16×16 bins that quantize entropy and the byte value. Before training, these counts are normalized to sum to unity [46].

String information The dataset includes 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, reported are 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. The printable characters distribution provides distinct information from the byte histogram information above since it is derived only from strings containing at least five consecutive printable characters. In addition, the string feature group includes the number of strings that begin with *C:* (case insensitive) that may indicate a path, the number of occurrences of *http://* or *http://* (case insensitive) that may indicate a *URL*, the number of occurrences of *HKEY_* that may indicate a registry key, and the number of occurrences of the short string *MZ* that may provide weak evidence of a Windows PE dropper or bundled executables. By providing a simple statistical summary of strings rather than a listing of raw strings, the authors mitigate privacy concerns that may exist for some benign files [46].

4.2 Sorel 20M Dataset

(Abstract) Sorel20M dataset is a large-scale dataset consisting of nearly 20 million files with pre-extracted features and metadata, high-quality labels derived from multiple source, information about vendor detections of the malware samples at the time of collection, and additional "tags" related to each malware sample to serve as additional targets. IN addition to features and metadata, the authors also provided approximately 10 million "disarmed" malware samples - samples with both the *optional_headers.subsystem* and *file_header.machine* flags set to zero - that may be used for further exploration of features and detection strategies [47].

(Introduction) The use of machine learning for malware detection is now relatively widespread. The ability for modern machine learning models to learn complex relationships between a large number of both statistical and parse-based features has lead to their widespread adoption. However, the risks associated with working directly with malware, as well as the commercial nature of much research in this space, has meant that most ML-based malware models are evaluated on private or proprietary datasets. This makes measuring progress in the field difficult. Furthermore, many sources of malware are commercial in nature, placing a high barrier of entry to the field and leading to researchers evaluating their models on extremely small datasets.

In contrast, fields like image classification or natural language processing have arguably benefited immensely from large, publicly available datasets such as CIFAR and ImageNet, which allow researchers to apply different approaches on a common dataset, making a direct comparison of those approaches possible. In addition to providing a basis for comparison between approaches, the existence of these common datasets has also made the fields more accessible, allowing smaller organizations that lacked the ability to compile large training and validation sets to contribute to the development of those fields.

The first attempt to address this lack was the seminal EMBER dataset [], which SOREL20M dataset builds upon. The EMBER dataset was the first standard dataset to be used for malware detection, however it had some shortcomings that limited its utility as a malware benchmark set. First, EMBER was of limited size, containing 900,000 training samples and 200,000 test samples, while commercial malware models are trained on tens to hundreds of millions of samples. In addition to the training size being too small to compare to commercial scale, the small validation size makes evaluation of model performance at lower false positive rates (1 in 1000 or below) difficult due to variance issues. Perhaps due to the relatively small size of the dataset, performance of classifiers on EMBER is nearly saturated, with a baseline classifier capable of obtaining an AUC of over 0.999. In addition, the EMBER dataset provided only pre-extracted features, making further research in such topic as improvements in feature extraction or realizable adversarial sample generation difficult. Finally, EMBER provides only a single binary label based on a simple "thresholding" rule.

SOREL-20M attempts to address these issues in whole or in part. The issue of training size is addressed by providing an order of magnitude more samples for analysis. Internally, the authors found that while performance continues to improve with larger datasets, validation sizes on the order of 3 to 4 million examples are sufficient to establish a stable rank order between models as well as to assess performance at lower false positive rates. 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.

The issue of feature exploration is partially addressed by providing (disarmed) binary samples for the malware only. The authors provided 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. They also have provided complete PE metadata as obtained via the Python *pefile* module using the *dump_dict* method.

Finally, the authors provided a number of additional targets (SMART tags) for the model that describe behaviour inferred from vendor labels [47].

The dataset is stored as a AWS bucket

4.2.1 Sorel 20M Dataset Description

The complete dataset consists of the following items:

- The 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 contains malware labels, tags, detection counts, and first/last seen times
  - The "ember_features" LMDB database contains EMBER features (extracted with version 2 of the features)
  - the "pe_metadata" LMDB database contains the PE metadata extracted via the *pefile* module, as described above

- 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; in the case of the disarmed malware samples, the sha256 of the original (unmodified) file was used, and not the sha256 of the disarmed file. The sha256 hash serves as the primary key for the SQLite3 database, and the key for 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. The authors suggested time-splits of the data - based upon the first-seen time in RL telemetry - as follows: 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 through the end of the data [47].

LMDB database, what is it? LMDB (Lightning Memory-Mapped Database) is a embedded database for key-value data based on B+trees. It is fully ACID transactional. The key features of LMDB are that it uses a single-level store based on memory-map files, which means that the OS is responsible for managing the pages (like caching frequently uses pages). It uses shared memory copy-on-write semantics with a single writer; readers do not block writers, writers do not block readers, and readers do not block readers. The system allows as many versions of data at any time as there are transactions (many read, one write). It also maintains a free list of pages to track and reuse pages instead of allocating memory each time.

—

LMDB is a Btree-based database management library modeled loosely on the BerkeleyDB API, but much simplified. The entire database is exposed in a memory map, and all data fetches return data directly from the mapped memory, so no malloc's or memcpy's occur during data fetches. As such, the library is extremely simple because it requires no page caching layer of its own, and it is extremely high performance and memory-efficient. It is also fully transactional with full ACID semantics, and when the memory map is read-only, the database integrity cannot be corrupted by stray pointer writes from application code.

The library is fully thread-aware and supports concurrent read/write access from multiple processes and threads. Data pages use a copy-on- write strategy so no active data pages are ever overwritten, which also provides resistance to corruption and eliminates the need of any special recovery procedures after a system crash. Writes are fully serialized; only one write transaction may be active at a time, which guarantees that writers can never deadlock. The database structure is multi-versioned so readers run with no locks; writers cannot block readers, and readers don't block writers.

Unlike other well-known database mechanisms which use either write-ahead transaction logs or append-only data writes, LMDB requires no maintenance during operation. Both write-ahead loggers and append-only databases require periodic checkpointing and/or compaction of their log or database files otherwise they grow without bound. LMDB tracks free pages within the database and re-uses them for new write operations, so the database size does not grow without bound in normal use.

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. Using read-write mode offers much higher write performance, but adds the possibility for stray application writes thru pointers to silently corrupt

the database. Of course if your application code is known to be bug-free (...) then this is not an issue [48].

—

Lightning Memory-Mapped Database (LMDB) is a software library that provides a high-performance embedded transactional database in the form of a key-value store. LMDB stores arbitrary key/data pairs as byte arrays, has a range-based search capability, supports multiple data items for a single key and has a special mode for appending records at the end of the database which gives a dramatic write performance increase over other similar stores. LMDB is not a relational database, it is strictly a key-value store.

LMDB may also be used concurrently in a multi-threaded or multi-processing environment, with read performance scaling linearly by design. LMDB databases may have only one writer at a time, however unlike many similar key-value databases, write transactions do not block readers, nor do readers block writers. LMDB is also unusual in that multiple applications on the same system may simultaneously open and use the same LMDB store, as a means to scale up performance. Also, LMDB does not require a transaction log (thereby increasing write performance by not needing to write data twice) because it maintains data integrity inherently by design.

4.2.2 Trying to improve Dataset Loading Speed

Dataset Pre-Processing

To speed up the code (particularly the trainig step) the dataset is pre-processed and saved to Google Drive in a easier (and faster) to read format. In particular the dataset is opened and read (and pre-processed), as was done in the original code every cycle, as it is saved 'as-is' to file. Doing so it is only needed to read it back into RAM 'as-is' during training, with no further processing. This greatly speeds up the overall training.

This however led to disk space problems since the pre-processed dataset is much bigger than the original one (which contained compressed features).

To speed up the code (specifically the training procedure, which is the one that takes longest), the dataset is pre-processed. Doing so speeded up the training execution: with the previous version I was able to train the model with a maximum of 400'000 samples in about 7:30h; now, instead, I managed to train the model with $2M$ samples in about 3:08h. This makes it feasable to run model training on $6M$ samples, which is a good amount of samples. However a new problem has now emerged: the pre-processed dataset occupies far more disk space than the original dataset (considering the same amount of samples). This is not a real problem for training, provided there is enough disk space for the pre-processed dataset alone (in fact I can pre-process the dataset separately and later import it without the need of downloding the original dataset).

I managed to pre-process $6M$ samples developing a variation of the pre-process function called "*pre-process multi*". In particular I pre-processed the dataset creating multiple small files which were saved on a different hard drive (google drive) than the instance one. Then, on a new (clean) instance I merged downloaded and merged all those files together, compressed the resulting pre-processed dataset and uploaded it on google drive. The final dataset size is approximately 70 GB (containing $6M$ samples), but after compression it takes about 18 GB of space. The dataset can now be imported on new instances and de-compressed before training. With this optimization I trained the model using $6M$ samples. The model took approximately 6 hours to complete just 1 training epoch. Considering 10 epochs I would need approximately 60 hours in total.

Alternative Generators (Dataloaders)

As a further optimization I tried implementing 2 new dataloader variants which could have been more efficient:

Generator alt1 The alternative generator 1 ('*alt1*') was inspired by the '*index select*' version of *FastTensorDataLoader* suggested in this post. In this version the dataset is loaded as a set of memory-mapped numpy arrays (one array for the features, one for the labels and one for the sha256 hashes) from which batches of data are randomly sampled using the pytorch '*index_select*' function. Moreover, my version (differently from the original suggested one) is able to work also with multiple workers (in *multithreading*).

Algorithm 1 Alt1 get_batch function

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

        if len(tensors) = 3 then
            batch[0] ← tensors[0][indices]
        end if

        for all t ∈ tensors[−2:] 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
```

Algorithm 2 Alt1 FastDataLoader class, Init

```
class FastTensorDataloader
    function Init(self, tensors, batch_size, shuffle, num_workers, ...)
        self.tensors ← tensors
        self.batch_size ← batch_size
        self.shuffle ← shuffle
        self.num_workers ← num_workers
        self.dataset_len ← tensors[0].shape[0]

        if num_workers > 1 then
            self.async_results ← [ ]
            self.pool ← ThreadPool()
        end if

        self.n_batches ← ⌈self.dataset_len / self.batch_size⌉
    end function
```

Generator alt2 The alternative generator 2 ('*alt2*') was inspired by the '*shuffle in-place*' version of *FastTensorDataLoader* suggested in this post. In this version the dataset is again loaded as a set of memory-mapped numpy arrays, as version 1. In this case, however, the arrays are randomly shuffled in place at each iteration, before sampling (in order) batches of data from them. Again, my version (differently from the original suggested one) is able to work also with multiple workers (in *multithreading*).

Algorithm 3 Alt1 FastDataLoader class, Iter

```
    function ITER(self)
        if self.shuffle = true then
            self.indices ← randperm(self.dataset_len)
        else
            self.indices ← None
        end if

        self.i ← 0
        return self
    end function
```

Algorithm 4 Alt1 FastDataLoader class, Next

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

        if self.num_workers = 1 then
            batch ← get_batch(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.num_workers do
                arguments ← (self.tensors, self.batch_size, self.i, self.indices)
                async_task ← self.pool.apply_async(get_batch, 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
end class
```

Generator alt3 Unfortunately, when considering $6M$ samples, the alternative generator version 1 ('*alt*1') was only slightly faster than the original Pytorch dataloader, while version 2 ('*alt*2') was considerably slower having to shuffle the entire memmap-ed dataset.

Since using a Google Cloud instance did not provide a significant speed-up, and since the model needed ~ 60 hours to complete 1 training run, I had to think of another dataloader optimization. I thus developed a further alternative dataloader ('*alt*3') that managed to reduce the time needed for 1 training epoch (considering 6M samples) from ~ 6 hours to ~ 15 minutes. The time needed to complete a full training run is thus now ~ 2.5 hours (~ 3 hours considering also model evaluation). The optimization consists of a tradeoff between loading speed and samples dispersion. In fact, the new dataloader uses a pool of threads to asynchronously load the dataset into memory in chunks. 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

Algorithm 5 Alt2 get_batch function

```
function GET_BATCH(tensors, batch_size, i, ...)
    batch ← [ ]
    for all t ∈ tensors do
        batch_data ← t[i:(i + batch_size)]
        batch.append(batch_data)
    end for

    return batch
end function
```

Algorithm 6 Alt2 FastDataLoader class, Init

```
class FastTensorDataloader
    function Init(self, tensors, batch_size, shuffle, num_workers, ...)
        self.tensors ← tensors
        self.batch_size ← batch_size
        self.shuffle ← shuffle
        self.num_workers ← num_workers
        self.dataset_len ← tensors[0].shape[0]

        if num_workers > 1 then
            self.async_results ← [ ]
            self.pool ← ThreadPool()
        end if

        self.n_batches ← ⌈self.dataset_len / self.batch_size⌉
    end function
```

Algorithm 7 Alt2 FastDataLoader class, Iter

```
    function ITER(self)
        if self.shuffle = true then
            r ← randperm(self.dataset_len)
            for i, t ∈ enumerate(self.tensors) do
                self.tensors[i] ← t[r]
            end for
        end if

        self.i ← 0
        return self
    end function
```

Algorithm 8 Alt2 FastDataLoader class, Next

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

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

of data at a time from it, when required. Selecting different values for *chunk_size* and *n_chunks* has an impact on the speed of the dataloader and on the samples dispersion. In fact the samples are not anymore randomly chosen from the whole dataset, but from a random sub-part of it. This 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 chosed carefully since their product results in the number of samples loaded into memory (RAM) at once for a single worker thread. Increasing too much this number (together with the number of workers used) will quickly saturate the available RAM.

Generator parameters optimization To understand the behaviour of the new dataloader and choose the best combination of values possible using 8 threads (which is double the amount of cores I have at my disposal in the Colab instance and I found is a good number of threads to use) I cross evaluated it using powers of 2 for both values. In particular I developed a function to be used to evaluate the new alternative generator on the target machine. This function performs 'epochs' model training epochs using values for *chunk_size* and *n_chunks* got from two intervals and plots 2 heatmaps: the first containing the average elapsed times and the second the average

Algorithm 9 Alt3 get_batch function

```
function GET_BATCH(tensors, batch_size, i, ...)
    batch ← [ ]
    for all t ∈ tensors do
        batch_data ← t[i:(i + batch_size)]
        batch.append(batch_data)
    end for

    return batch
end function
```

Algorithm 10 Alt3 get_chunks function

```
function GET_CHUNKS(tensors, chunk_indices, chunk_size, last_chunk_size, n_chunks, shuffle)
    if n_chunks −1 in chunk_indices then
        chunk_agg_size ← (len(chunk_indices) −1) · chunk_size + last_chunk_size
    else
        chunk_agg_size ← len(chunk_indices) · chunk_size
    end if

    chunk_agg ← [ ]
    if len(tensors) = 3 then
        chunk_agg.append(emptyNumpyArray)
        for all t ∈ tensors[−2:] do
            chunk_agg.append(emptyTensor)
        end for
    else
        for all t ∈ tensors do
            chunk_agg.append(emptyTensor)
        end for
    end if

    c_start ← 0
    for all idx ∈ [0, ..., len(chunk_indices)] do
        t_start ← chunk_indices[idx] · chunk_size
        if chunk_indices[idx] ≠ n_chunks −1 then
            c_end ← c_start + chunk_size
            t_end ← t_start + chunk_size
        else
            c_end ← c_start + last_chunk_size
            t_end ← t_start + last_chunk_size
        end if
        for i, t ∈ enumerate(tensors) do
            chunk_agg[i][c_start:c_end] ← t[t_start:t_end]
        end for
        c_start = c_end
    end for

    if shuffle = true then
        r = randperm(chunk_agg_size)
        for i, t ∈ enumerate(chunk_agg) do
            chunk_agg[i] = t[r]
        end for
    end if

    return chunk_agg, chunk_agg_size
end function
```

Algorithm 11 Alt3 FastDataLoader class, Init

```
class FastTensorDataloader
    function Init(self, tensors, batch_size, chunk_size, chunks, shuffle, num_workers, ...)
        self.tensors ← tensors
        self.batch_size ← batch_size
        self.chunk_size ← chunk_size
        self.chunks ← chunks
        self.shuffle ← shuffle
        self.num_workers ← num_workers
        self.dataset_len ← tensors[0].shape[0]

        if num_workers > 1 then
            self.async_results ← [ ]
            self.pool ← ThreadPool()
        end if

        self.n_batches ← ⌈self.dataset_len / self.batch_size⌉
        self.n_chunks ← ⌈self.dataset_len / self.chunk_size⌉
        self.last_chunk_size ← self.dataset_len % self.chunk_size
    end function
```

Algorithm 12 Alt3 FastDataLoader 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
```

speeds. In my evaluation run I hard-coded for $chunk\_size$ the powers of 2 ranging from 2^4 to 2^14 (included) and for n_chunks the powers of 2 ranging from 2^3 to 2^13. Moreover, two further parameters have to be set in order to constrain the evaluation to meaningful values: 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 $chunk\_aggregate$ (which resides in main memory). min_mul and max_mul indicate the minimum and maximum number of batches retrievable from the resulting $chunk\_aggregate$. In my run I considered for min_mul the value 1 and for max_mul the value 32. This was done in order to be able to retrieve at least one batch from the $chunk\_aggregate$ without saturating the available RAM (having for max_mul a value greater than $32 * batch\_size$ would be too mutch for my instance). Given the results I got (available at the end of this document), I decided to stick with the following values for all the model training runs since they seemed to be a good compromise between loading speed and samples dispersion: $chunk\_size = 256$ and $n\_chunks = 256$.

4.3 Fresh Dataset

To properly evaluate the model there is the need of a dataset with malware family labels. Malware Bazaar provides examples of malware executables for different malware families and it could therefore be a good source for creating a new (fresh) dataset. I therefore created a function which, given:

- a series of malware families in order of importance
- the number 'x' (default: 100) of samples per family to download
- the number 'm' (default: 10) of families to consider

downloads from Malware Bazaar 'x' samples per malware family (if possible, otherwise it skips the malware family and considers the next), extracts the features from each sample and creates the new dataset (which I call '$fresh\_dataset$' in the code) with '$n = x * m$' (default: 1000) samples compatible with the model.

4.3.1 Model Evaluation with Fresh Dataset

I implemented a model evaluation function using the $fresh\_dataset$. This function randomly samples 'q' (default: 100) query samples among the ones in the $fresh\_dataset$, it computes the similarity between each of these samples with the other '$n - 1$' (default: 999), and orders these '$n - 1$' samples by similarity. It then calcluates the MRR (Mean Reciprocal Rank) and MAP (Mean Average Precision) scores for the model. The rankings for the 'q' (100) query samples are saved as a single json file. Moreover, 5 particular rankings are saved as csv files. These 5 rankings are those that produced:

Algorithm 13 Alt3 FastDataLoader class, Next

```
    function NEXT(self)
        if self.num_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
                self.chunk_agg, self.chunk_agg_size ← get_chunks(self.tensor, self.chunk_indices[start_i:end_i], self.chunk_size, self.last_chunk_size, self.n_chunks, self.shuffle)
                self.chunk_i ← end_i
                self.i ← 0
            end if
        else
            while self.chunk_i < self.n_chunks and len(self.async_results) < self.num_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(get_chunks, 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 ← get_batch(self.chunk_agg, self.batch_size, self.i)
        self.i ← self.i + batch[0].shape[0]
        return batch
    end function
end class
```

- the maximum RR (Reciprocal Rank)
- the minimum RR (Reciprocal Rank)
- the maximum AP (Average Precision)
- the minimum AP (Average Precision)
- the last ranking is randomly sampled between all the other rankings

The model fresh dataset evaluation computes both the MRR and MAP scores. The Mean Reciprocal Rank (MRR) is the average of the Reciprocal Ranks (RR) of a series of queries. 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 calssified as closest (highest rank). The MAP score instead takes into account, with proper weights, all the sampels in the family of interest. Therefore a model will have a higher MRR score if it classifies a single sample belonging to the family of interest higher in the ranking; and it will have a higher MAP score if it classifies all the samples belonging to the family of interest higher in the ranking.

Chapter 5

Experimenting with ML based Malware Detection/Description methods

Chapter 6

Proposed Tool

Description of the proposed tool..

Chapter 7

Experiments with the proposed tool

Chapter 8

Results

Results analysis..

Chapter 9

Conclusions

Qui si inseriscono brevi conclusioni sul lavoro svolto, senza ripetere inutilmente il sommario.

Si possono evidenziare i punti di forza e quelli di debolezza, nonché i possibili sviluppi futuri o attività da svolgere per migliorare i risultati.

Chapter 10

Appendix

10.1 Notable Examples of Malware in Recent History

Here is an overview of the most famous malware or malware-related events in recent history:

- ***Melissa*** (1999) - Considered to be one of the first cases of social engineering in history, the Melissa mass-mailing macro virus infected thousands of computers worldwide by the end of 1999. The virus was spread via e-mail, using a malicious Word attachment named "list.doc" and as subject: "Important message from", followed by the victim's username. Once opened, the attachment would execute a macro that mass-mailed the virus to the first 50 people in the user's contact list and disabled multiple security features on Microsoft Word and Microsoft Outlook.
- ***ILOVEYOU*** (2000) - *ILOVEYOU*, sometimes referred to as *Love Bug* or *Love Letter for you*, was a computer worm that infected over ten million Windows personal computers in 2000. It spread as an email message with the subject line "ILOVEYOU" and the attachment "LOVE-LETTER-FOR-YOU.txt.vbs". Opening the attachment would activate the Visual Basic script which caused damages on the local machine, overwriting random files. Finally, the worm sent a copy of itself to all addresses in the Windows Address Book used by Microsoft Outlook.
- ***SQL Slammer*** (2003) - This malware exploited a buffer overflow bug in Microsoft's SQL Server and Desktop Engine database products, causing a denial of service on some Internet hosts and dramatically slowing general Internet traffic. It spread rapidly, infecting most of its 75,000 victims within ten minutes.
- ***MyDoom*** (2004) - Also known as *W32.MyDoom@mm*, *Novarg*, *Mimail.R* and *Shimgapi*, it was a computer worm which affected Microsoft Windows systems. It became known mostly because it tried hitting major technology companies, such as Google and Microsoft. It spread by email using attention-grabbing subjects, such as ?Error?, ?Test? and ?Mail Delivery System?. It became the fastest-spreading e-mail worm ever, exceeding previous records set by the Sobig worm and ILOVEYOU, a record which as of 2021 has yet to be surpassed.
- ***Conficker*** (2008) - Also known as *Downup*, *Downadup* and *Kido*, this computer worm targeted the Microsoft Windows operating system. It used flaws in Windows OS software and dictionary attacks on administrator passwords to propagate while forming a botnet. The Conficker worm infected over 15 million Windows systems including government, business and home computers in more than 190 countries. This made it the largest known computer worm infection since the 2003 *Welchia*.
- ***Zeus*** (2007-2009) - Also known as *ZeuS*, or *Zbot*, *Zeus* was a Trojan horse malware that ran on Microsoft Windows. It was able to carry out many malicious and criminal tasks.

However, it was most often used to steal banking information through the use of man-in-the-browser keystroke logging and form grabbing. It was also often used to install the *CryptoLocker* ransomware. Zeus spread mainly through drive-by downloads and phishing mails. First identified in mid 2007, it became more widespread in 2009.

- ***Stuxnet Worm*** (2010) - *Stuxnet* was an extremely sophisticated worm that infected computers worldwide. It was allegedly developed by US and Israeli intelligence to hinder the Iranian nuclear program. It was introduced into the target environment (Iran's nuclear power plant) via a flash drive. Stuxnet's escape from the target environment, which was air-gapped, was not expected. Once in the wild, Stuxnet spread aggressively but mostly harmed the target Iranian nuclear facility, where it damaged uranium-enrichment centrifuges, causing little damage outside of its intended target environment [13].

- ***CryptoLocker*** (2013) - It is considered to be one of the first widespread ransomware attacks. It targeted computers running Microsoft Windows and it propagated via infected email attachments. When activated, it encrypted certain types of files stored on local and mounted network drives using RSA public-key cryptography. The private key was stored only on the malware's control servers. A message was then displayed offering to decrypt the data if a payment (through either bitcoin or other means) was made before a certain deadline. After such deadline the private key was deleted. However, paying the ransom did not always lead to the files being decrypted.

  Its code now keeps getting repurposed in similar malware projects.

- ***Mirai*** (2016) - *Mirai* is an infamous malware which compromised vulnerable IoT (Internet of Things) devices - such as IP cameras and home routers - turning them into remotely controlled bots.

  The Mirai botnet, one of the biggest (and worse) botnets in existence, was first found in August 2016 and has been used in some of the largest and most disruptive distributed denial of service (DDoS) attacks, including an attack on computer security journalist Brian Krebs' web site. The source code for Mirai has been published on Hack Forums as open-source, as a result its techniques have been adapted in many other malware projects.

- ***Petya*** *and* ***NotPetya*** (2016-2017) - These malware attacks spread globally, however their damages particularly targeted Ukraine, where the national bank was hit. The Petya ransomware family caused an estimated $10 billion in damages worldwide [17]. *Petya* targeted Microsoft Windows-based systems, infecting the Master Boot Record (MBR) to execute a payload that encrypted the hard drive's file system table preventing Windows from booting. It subsequently demanded a ransom in Bitcoin to the user in order to regain access to the system.

  Many variants of Petya were created in the subsequent months. In mid 2017, a new variant of Petya was used for a global cyber-attack, again primarily targeting Ukraine. The new variant spread using the EternalBlue exploit, the same one used by the WannaCry ransomware. This new version was called *NotPetya* to distinguish it from the 2016 variants.

- ***WannaCry*** (2017) - WannaCry is considered to be one of largest ransomware attacks in history. It targeted computers running Microsoft Windows by encrypting data and demanding ransom payments in Bitcoin. It mainly propagated through EternalBlue, an exploit developed by the U.S. National Security Agency (NSA) for older Windows systems. This exploit was stolen from NSA and leaked approximately one year before the attack. While Microsoft had released security patches against this exploit, some organizations had not applied them, or were using older Windows systems when the attack occurred.

  The attack was stopped within a few days of its discovery thanks to emergency patches released by Microsoft and the discovery of a kill switch. Before being stopped, WannaCry was spread infecting systems at a terrifying rate of 10,000 PCs per hour [17]. In the end, the attack was estimated to have affected more than 200,000 computers across 150 countries, with total damages ranging from hundreds of millions to billions of dollars.

- ***Emotet*** (2018) - This malware, also known as *Heodo*, was first detected in 2014 as a banking trojan aimed at stealing banking credentials from infected hosts. Throughout 2016

and 2017, its creators updated and reconfigured it to work primarily as a "loader" - a type of malware that gains access to a system, and then allows its operators to download and execute additional payloads. These payloads can be any type of executable code, from Emotet's own modules to malware developed by other cybercriminals.

This malware usually makes its way on target systems via a macro virus attached to an email. The infected email appears to be a legitimate reply to an earlier message sent by the victim. When on the system, it is particularly difficult to combat because it evades signature-based detection, is persistent, and includes advanced spreader modules used to propagate effectively. Emotet authors have often used the malware to create a botnet of infected computers to which they sell access in Malware-as-a-Service to other cybercriminals, such as the Ryuk gang.

In 2020, Emotet spread again globally, infecting its victims with TrickBot and Qbot, which are used to steal banking credentials and spread inside networks. In January 2021, Europol and Eurojust coordinated international actions allowed investigators to take control of and disrupt the Emotet infrastructure.

- ***COVID-19 related attacks*** (2020) - In 2020, many cybercriminals shamelessly took advantage of the people's fear of coronavirus during the COVID-19 pandemic through COVID-19 related phishing scams. Using fake communications, for example spoofing the World Health Organization, attackers deployed malware and got access to targets' sensitive information among other nefarious actions [17].

  Another COVID-19 attack was that of a malicious Android app called *CovidLock*, which claimed to be a real-time coronavirus outbreak tracker but instead was a ransomware that attempted to trick the user into providing administrative access on their device and then locked it requesting a ransom.

Bibliography

[1] 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

[2] R. Sharp, "An introduction to malware." https://orbit.dtu.dk/en/publications/an-introduction-to-malware, 2017

[3] 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

[4] NIST, "malware." https://csrc.nist.gov/glossary/term/malware, Accessed: 2021-03-15

[5] 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

[6] Symantec, "Difference between viruses, worms, and trojans." https://knowledge.broadcom.com/external/article?legacyId=TECH98539, 2019, Accessed: 2021-03-15

[7] 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

[8] 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

[9] 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

[10] N. DuPaul, "Common malware types: Cybersecurity 101." https://www.veracode.com/blog/2012/10/common-malware-types-cybersecurity-101, 2012, Accessed: 2021-03-15

[11] S. Ingalls, "Types of malware and best malware protection practices." https://www.esecurityplanet.com/threats/malware-types/, 2021, Accessed: 2021-03-15

[12] MyraSecurity, "What is malware?." https://www.myrasecurity.com/en/what-is-malware/, Accessed: 2021-03-15

[13] K. Baker, "The 11 most common types of malware." https://www.crowdstrike.com/cybersecurity-101/malware/types-of-malware/, 2021, Accessed: 2021-03-15

[14] T. Femister, "Encryption happens last: The ransomware revolution." https://www.forbes.com/sites/forbestechcouncil/2020/08/18/encryption-happens-last-the-ransomware-revolution/?sh=4c267d34414b, 2020, Accessed: 2021-03-15

[15] McAfee, "What is malware?." https://www.mcafee.com/en-us/antivirus/malware.html, Accessed: 2021-03-15

[16] 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

[17] 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

[18] B. Lutkevich, "malware." https://searchsecurity.techtarget.com/definition/malware, Accessed: 2021-03-15

[19] P. Szor, "The art of computer virus and defence", Symantec press, 1st ed., 2005, ISBN: 978-0-321-30454-4

[20] 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

[21] E. Skoudis and L. Zeltser, "Malware: Fighting malicious code", Prentice Hall Professional, 2004, ISBN: 978-0-131-01405-3

[22] E. Eilam, "Reversing: Secrets of reverse engineering", John Wiley &; Sons, Inc., 2005, ISBN: 9780764574818

[23] 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

[24] 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

[25] 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

[26] 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

[27] 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

[28] A. Balakrishnan and C. Schulze, "Code obfuscation literature survey." http://pages.cs.wisc.edu/~arinib/writeup.pdf, 2005

[29] I. You and K. Yim, "Malware obfuscation techniques: A brief survey", 11 2010, pp. 297–300, DOI 10.1109/BWCCA.2010.85

[30] 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

[31] 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

[32] 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

[33] 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

[34] V. Nguyen, "A study of polymorphic virus detection", 11 2018, DOI 10.13140/RG.2.2.19853.79842

[35] 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

[36] 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

[37] 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

[38] 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

[39] Y. Miao, "Understanding heuristic-based scanning vs. sandboxing." https://www.opswat.com/blog/understanding-heuristic-based-scanning-vs-sandboxing, 2015, Accessed: 2021-06-13

[40] Kaspersky, "What is heuristic analysis?." https://usa.kaspersky.com/resource-center/definitions/heuristic-analysis, Accessed: 2021-06-13

[41] Forcepoint, "What is heuristic analysis?." https://www.forcepoint.com/cyber-edu/heuristic-analysis, Accessed: 2021-06-13

[42] A. Kyadige, E. M. Rudd, and K. Berlin, "Learning from context: Exploiting and interpreting file path information for better malware detection", 2019
[43] E. M. Rudd, F. N. Ducau, C. Wild, K. Berlin, and R. Harang, "Aloha: Auxiliary loss optimization for hypothesis augmentation", 2019
[44] 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
[45] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features", 2015
[46] H. S. Anderson and P. Roth, "Ember: An open dataset for training static pe malware machine learning models", 2018
[47] R. Harang and E. M. Rudd, "Sorel-20m: A large scale benchmark dataset for malicious pe detection", 2020
[48] H. Chu, "Lightning memory-mapped database manager (lmdb) documentation." http://www.lmdb.tech/doc/, Accessed: 2021-06-22