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, malware detection with machine learning techniques is gaining popularity. Tom Mitchell [] defines machine learning as the study of computer algorithms that improve through experiments. Robert Moskovitch et. al. [] proposed detection of malwares based on monitoring the computer behaviour (features). His evaluation results suggest that by using classification algorithms applied on only 20 features the mean detection accuracy exceeded 90%. The advantage of machine learning techniques is that it will not only detect known malwares but also act as knowledge for the detection of new malware. The popular machine learning techniques among the researchers for the detection of second generation malwares are Naive Bayes, Decision Trees, Data Mining, Neural Networks and Hidden Markov Modes. This technique may not replace the standard detection methods, but it can act as an add-on feature. Generally, machine learning techniques are more computationally demanding than the standard anti-malware, hence it may not be suitable for end users. However, it can be implemented at enterprise gateaway level to act as a central anti-malware engine to supplement anti-malwares. Although infrastructure requirement is costly, it can help in protecting valuable enterprises data from the security threats and can prevent immense financial damages [20].

In recent years, machine learning has gained its popularity in many fields including IT security. Robert Moskovitch et. al. [] proposed a technique that monitors a small set of features that are sufficient for detecting malware without sacrificing accuracy. The result of the study showed that, using only 20 features, the mean detection accuracy was greater than 90%, and for specific unknown worms, this accuracy got over 99%, while maintaining a low level of false positives. The advantage of machine learning techniques is that it will not only detect a known malware but also act as a database for detecting new malware. Similar studies can also be found in another model such as Naive Bayes, Decision Trees, Neural Networks. Although this technique is practical, it may not replace the standard detection methods, rather than act as an add-on feature because machine learning techniques are computationally demanding and may not be suitable for end users [34].

3.6.1 AlOHA: Auxiliary Loss Optimization for Hypothesis Augmentation

Rudd et al. introduced several loss functions, including a multi-target binary cross entropy loss over multiple malicious/benign detection sources, a Poisson loss over total detection counts from all malicious/benign sources, and a multi-target binary cross entropy loss over semantic malware attribute tags. The auxiliary labels provide some contextual information to guide the training process [42].

—

(Abstract) Malware detection is a popular application of Machine Learning for Information Security (ML-Sec), in which a ML classifier is trained to predict whether a given file is malware of benignware. Parameters of this classifier are typically optimized such that outputs from the model over a set of input samples most closely match the samples true malicious/benign (1/0) target labels. However, there are often a number of other sources of contextual metadata for each malware sample, beyond an aggregate malicious/benign label, including multiple labelling sources and malware type information (e.g. *ransomware*, *trojan*, etc.), which we can feed to the classifier as auxiliary prediction targets. The authors fit deep neural networks to multiple additional targets derived from metadata in a threat intelligence feed for Portable Executable (PE) malware and benignware, including a multi-source malicious/benign loss, a count loss on multi-source detections, and a semantic malware attribute tag loss. They found that incorporating multiple auxiliary loss terms yielded a marked improvement in performance on the main detection task. Those gains likely stem from a more informed neural network representation and are not due to a regularization artefact of multi-target learning [43].

—

(Introduction) Machine learning (ML) for computer security (ML-Sec) has proven to be a powerful tool for malware detection. ML models are now integral parts of commercial anti-malware engines and multiple vendors in the industry have dedicated ML-Sec teams. For the malware detection problem, these models are typically tuned to predict a binary label (malicious or benign) using features extracted from sample files. Unlike signature engines, where the aim is to reactively blacklist/whitelist samples that hard-match manually-defined patterns (signatures), ML engines employ numerical optimization on parameters of highly parametrized models that aim to learn more general concepts of *malware* and *benignware*. This allows some degree of proactive detection of previously unseen malware samples that is not typically provided by signature-only engines.

Frequently, malware classification is framed as a binary classification task using a simple binary cross-entropy or two-class softmax loss function. However, there often exist substantial metadata available at training time that contain more information about each input sample than just aggregate label of whether it is malicious or benign. Such metadata might include malicious/benign labels from multiple sources (e.g. from various security vendors), malware family information, file attributes, temporal information, geographical location information, counts of affected endpoints, and associated tags. In many cases this metadata will not be available when the model is deployed, and so in general it is difficult to include this data as features in the model.

It is popular practice in the domain of malware analysis to derive binary malicious/benign labels based on a heuristic combination of multiple detection sources for a given file, and then use these noisy labels for training ML models. However, there is nothing that precludes training a classifier to predict each of these source labels simultaneously optimizing classifier parameters over these predictions + labels. In fact, guiding a model to develop representations capable of predicting multiple targets simultaneously may have a smoothing or regularizing effect conductinve to generalization, particularly if the auxiliary targets are related to the main target of interest. These auxiliary targets can be ignored during test time if they are ancillary to the task at hand (and in many cases the extra weights required to produce them can be removed from the model prior to deployment), but nevertheless, there is much reason to believe that forcing the model to fit multiple targets simultaneously can improve performance on the key output of interest. The authors took advantage of multi-target learning by exploring the use of metadata from threat intelligence feeds as auxiliary targets for model training.

The authors studied both the addition of multiple label sources for the same task and multiple label sources for multiple separate tasks. They did not presume the presence of all labels from all sources, and introduced a per-sample weighting scheme on each loss term to accommodate missing labels in the metadata. They further explored the use of multi-objective training as a way to expand the number of trainable samples in cases where the aggregate malicious/benign label is unclear, and where those samples would otherwise be excluded from purely binary training [43].

—

(ML-Sec Detection Pipelines: From Single Objective to Multi-Objective) The authors described a simplified ML-Sec pipeline for training a malicious file classifier, and proposed a simple extension that allowed the use of metadata available during training (but not at test time) and improves performance on the classification task.

ML-Sec detection pipelines use powerful machine learning models that require many labelled malicious and benign samples to train. The exact nature of the metadata varies, but typically, malicious and benign scores are provided for each of M individual samples on a per-vendor basis, such that, given V vendors, between 0 and V of them will designate a sample malicious. For a given sample, some vendors may choose not to answer, resulting in a missing label for that vendor. Furthermore, many vendors also provide a detection name (or malware family name) when they issue a detection for a given file. Additional information may also be available, but crucially, the following metadata are presumed present for the models presented by the authors:

1. per-vendor labels for each sample, either malicious/benign (mapped to binary I/O, respectively) or *NULL*
2. textual per-vendor labels on the sample describing the family and variant of the malware (an empty string if benign or *NULL*)
3. time at which the sample was first seen

Using the individual vendor detections, an aggregate label can be derived either by a voting mechanism or by thresholding the net number of vendors that identify a given sample as malicious.

Each malware/benignware sample must also be converted to a numerical vector to be able to train a classifier, a process called *feature extraction*. The authors focused on static malware detection, meaning that they assume only access to the binary file, as opposed to dynamic detection, in which the features used predominantly come from the execution of the file. The feature extraction mechanism consists of some numerical transformation that preserves aggregate and fine-grained information throughout each sample, for example, the feature extraction proposed by Saxe et al. [] uses windowed byte statistics, 2D histograms of delimited string hash vs. length, and histograms of hashes of PE-format specific metadata - e.g. imports from the import address table (IAT).

Given extracted features and derived labels, a classifier is then trained, tuning parameters to minimize misclassification as measured by some loss criterion, which under the constraints

of a statistical noise model measures the deviation of predictions from their ground truth. For both neural networks and ensemble methods a logistic sigmoid is commonly used to constrain predictions to a [0,1] range, and a cross-entropy loss between predictions and labels is used as the minimization criterion under a Bernoulli noise model per-label.

Note that much information in the metadata, which is often not used to determine the sample label but *is* correlated to the aggregate classification, is not used in training, e.g., the individual vendor classifications, the combined number of detections across al vendors, and information related to malware type that could be derived from the detection names. The authors augment a deep neural network malicious/benign classifier with additional output predictions of vendor counts, individual vendor labels, and/or attribute tags. These separate prediction arms were given their own loss functions which they jointly minimized through back-propagation [43].

—

Implementation Details The model uses various loss functions - denoted by $L_{loss\ type}(X, Y)$ for some input features X and targets Y - associated with the various outputs of the model, as well as how the labels Y representing the targets of these outputs are constructed.

The base for our model is a feedforward neural network incorporating multiple blocks composed of Dropout, a dense layer, batch normalization, and an exponential linear unit (ELU) activation. The core of the model contains five such blocks with 1024, 768, 512, 512, and 512 hidden units, respectively. This base topology applies the function $f()$ to the input vector to produce an intermediate 512 dimensional representation of the input file $\mathbf{h} = f(\mathbf{x})$. They then append to this model an additional block, consisting of one or more dense layers and activation functions, for each output of the model. They denote the composition of their base topology and their target-specific final dense layers and activations applied to features $\mathbf{x}$ by $f_{target}(\mathbf{x})$. The output for the main malware/benign prediction task - $f_{mal}(\mathbf{x})$ - is always present and consists of a single dense layer followed by a sigmoid activation on top of the base shared network that aims to estimate the probability of the input sample being malicious. A network architecture with only this malware/benign output serves as the baseline for our evaluations. To this baseline model they add auxiliary outputs with similar structure as described above: one fully connected layer (two for the *tag* prediction task) which produces some task-specific number of outputs (a single output, with the exception of the restricted generalized Poisson distribution output, which uses two) and some task-specific activation. Except where noted otherwise, all multi-task losses were 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). Training was standardized to 10 epochs [43].

—

Malware Loss For the task of predicting if a given binary file, represented by its features $\mathbf{x}^i$, is malicious or benign we used a 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:

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

The authors used a "1-/5+" criterion for labelling a given file as malicious or benign: if a file has one or fewer vendors reporting it as malicious, they label the file as 'benign' and use a weight of 1.0 for the malware loss for that sample. Similarly, if a sample has five or more vendors reporting it as malicious, they label the file as 'malicious' and use a weight of 1.0 for the malware loss for that sample [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 $a(\dot{})$ 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:

$$\begin{aligned} L_{vdr}(X, Y) &= \frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{V} l_{vdr}(f_{vdr_j}(\mathbf{x}^{(i)}), y_{v_j^{(i)}}) \\ &= -\frac{1}{M} \sum_{i=0}^{M} \sum_{j=1}^{V} y_{v_j}^{(i)} \log(\hat{y}_{v_y}^{(i)}) + (1 - y_{v_j}^{(i)}) \log(1 - \hat{y}_{v_j}^{(i)}), \end{aligned} \tag{3.6}$$

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, for example, the token *xmrig* - even though it is not a malware family - can be recognised as referring to a crypto-currency mining software and therefore can be associated with the *crypto-miner* tag. The authors then created a mapping from tokens to tags based on prior knowledge. They labelled a sample as associated with tag t_i if any of the tokens associated with t_i are present in any of the detection names assigned to the sample by the set of trusted vendors.

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. The vendor counts and the individual vendor labels are closely related with the definition of their main target, i.e. the malicious label, which classifies a sample as malicious if 5 or more vendors identify the sample as malware. In the case of tag targets, this information is not present. For instance, if all the vendors recognize a given sample as coming from the *WannaCry* family in their detection names, the sample will be associated only once with the *ransomware* tag. ON the converse, because of their tagging mechanism, if only one vendor considers that a given sample is malicious and classifies it as coming from the *WannaCry* family, the *ransomware* tag will be present (although the malicious label will be 0).

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:

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

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.

Sample Weights While the multi-objective network has the advantage that multiple labels and loss functions serve as additional sources of information, this introduces an additional complexity: given many (potentially missing) labels for each sample, we cannot rely on having all labels for a large quantity of the samples. To address this, the authors incorporated per-sample weights, depending on the presence and absence of each label. For labels that are missing, they assigned them a default value and then set the associated weights to zero in the loss computation so a sample with a missing target label will not add to the loss computation for that target. This allows to train the network, even in the presence of partially labelled samples [43].

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 did not use a randomized cross-validation training/test split as is common in other fields, because that would allow the set on which the classifier was trained to contain files "from the future", leading to spuriously optimistic results. The reason for the one month gap between the end of the training/validation set and the start of the test set is to simulate a realistic worst-case deployment scenario where the detection model of interest is updated on a monthly basis.

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.

Note that ROC curves, which are commonly used as performance measures, are independent of class ration in the test set (unlike accuracy), since false positive rate (FPR) values depend only on the benign data, and true positive rate (TPR) values depend only on malware. 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

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

As the training process for deep for deep neural networks has some degree of intrinsic randomness, which can result in variations in their performance, we report our results in terms of both the mean and standard deviation for the test statistics of interest across five runs. Each model was trained five times, each time with a different random initialization and different randomization over mini-batches, and all other parameters held identical. The authors computed the test statistic of interest (e.g. the detection rate at a false positive rate of 10^{-3}) for each model, and then computed the average and standard deviation of those results.

Related Work Applications of ML to computer security date back to the 1990's, but large-scale commercial deployments of deep neural networks (DNNs) that have led to transformative performance gains are a more recent phenomenon. Several works from the ML-Sec community have leveraged DNNs for statically detecting malicious content across a variety of different formats and file types. However, these works predominantly focus on applying regularized cross-entropy loss functions between single network outputs and malicious/benign labels per-sample, leaving the potential of multiple-objective optimization largely untapped.

A notable exception is [], in which Huang et al. add a multiclass label for Microsoft's malware families to their classification model using a categorical cross-entropy loss function atop a softmax output as an auxiliary objective. They observed that adding targets in this fashion increased performance both on the detection task and on the malware family classification task. While their work used 4000-dimensional dynamic features derived from Windows API calls, ALOHA authors extended multi-target approaches to lower-dimensional static features on a larger data set (more scalable approach however lacking the advantages of dynamic features).

Despite the lack of attention from the ML-Sec community, multi-target learning has been applied to other areas of ML for a long time. Caruna [] first introduced multi-task learning in neural networks as a "source of *inductive bias*" (also referred to as '*inductive transfer*'), in which more difficult tasks could be combined in order to exploit similarities between tasks that could serve as complementary signals during training. Caruna demonstrated that jointly learning related tasks produces better generalization on a task-by-task basis than learning them individually. Moreover, he also demonstrated that learning multiple copies of the same task can also lead to a modest improvement in performance.

Kumar and Duame [] considered a refinement on the basic multi-task learning approach that led to clustering related tasks, in an effort to mitigate the potential of *negative transfer* in which unrelated tasks degrade performance on the target task.

Multi-target learning has been applied to extremely complex image classification tasks, including predicting characters and n-grams within unconstrained images of text, joint facial landmark localization and detection, image tagging and retrieval, and attribute prediction where a common auxiliary task is to challenge the network to classify additional attributes of the image, such as manner of dress for full-body images of people or facial attributes.

The widespread success of transfer learning is also a testament to the value of training a single model on nominally distinct tasks.

Multi-view learning is a related approach in which multiple inputs are trained to a single target. This approach also arguably leads to the same general mechanism for improvement: the model is forced to learn relationships between sets of features that improve the performance using any particular set. While this approach often requires all sets of features to be available at test time, there are other approaches, such as [], that relax this constraint.

3.6.2 Automatic Malware Description via Attribute Tagging and Similarity Embedding

(Abstract) With the rapid proliferation and increased sophistication of malicious software (malware), detection methods no longer rely only on manually generated signatures but have also incorporated more general approaches like machine learning detection. Although powerful for conviction of malicious artefacts, these methods do not produce any furhter 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

Semantic Attribute Tagging

Multi-Label Classification

Malware Analysis with Neural Networks

Semantic Malware Attribute Tags

Tag Distillation from Detection Names

Tags Prediction - Implementation Details

Evaluation of Tagging Algorithms

Dataset and Training details

Latent Space (Joint Embedding) Analysis

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 mischaracterised 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) Commercial models typically rely on deep neural networks, or boosted decision tree ensembles and have been extended to other static file types as well, including web content, office documents, and archives. While

methods for dealing with these different input types have their own intricacies, they typically use single inputs derived from file content as a feature vector or text embedding.

Static ML detectors use highly parametric classifiers trained on many malicious and benign samples. The goal is to tune the parameters of these classifiers to best match the outputs from the classifiers for all input samples to their actual ground truth labels. Provided that the malware/benignware samples in the training set are similar enough in content to those seen at deployment and that the samples are well labelled, the learned detection function should work well.

In practice, labels are often collected from vendor aggregation feeds, which submit samples to malware detectors from a variety of vendors. The results can be aggregated into labels that are usually correct, e.g. by using a 1-/5+ criterion or treating the label as a hidden variable and using statistical estimation methods. Often a time lag is introduced to let vendors update their models, blacklists, and whitelists accordingly. Generally, the longer the time lag, the more accurate the labels, but the less the data resembles that of the deployment distribution. In actual deployment contexts, classifiers are retrained on new data/labels periodically and the updated parameters are sent to the endpoints on which the detectors are running.

Most static ML for information security (ML-Sec) classifiers operate on learned embeddings over portions of files (e.g. headers), learned embeddings over the full file, or most commonly, on pre-engineered numerical *feature vectors* designed to summarize the content from each file. Learned embeddings, which generally presume some sort of convolutional architecture, have the advantage that they do not presume a fixed structure and are derived directly during training. However, this process is significantly more expensive, and does not scale as gracefully, e.g., to tens to hundreds of millions of large PE files. Moreover generic bytes are inherently less constrained than inputs like images, video, audio, and text, where convolutions can take advantage of structural localities/hierarchies. Thus, for generic malicious/benign files there is less performance benefit from learning to embed features directly from inputs. Pre-engineered feature vector representations, by contrast quickly distil content from each file that is informative in a classification sense. There are a number of ways to craft feature vectors, including tracking per-byte statistics over sliding windows, byte histograms, n-gram histograms, treating bytes as pixel values in an image (a visualisation of the file content), opcode and function call graph statistics, symbol statistics, hashed/numerical metadata values - e.g. entry point as a fraction of the file, or hashed imports and exports, - and hashes of delimited tokens. In practical applications, several different types of feature vectors extracted from file content are often concatenated together to achieve superior performance [42].

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:

$$L(\mathbf{x}, y; \theta) = -y \log\left(f(\mathbf{x};\theta)\right) + (1-y)\log\left(1 - f(\mathbf{x};\theta)\right). \tag{3.9}$$

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

$$\hat{\theta} = \arg\max_{\theta} \sum_{i=1}^{M} L(\mathbf{x}^{(i)}, y_i; \theta), \tag{3.10}$$

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

Dataset Used

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