Endless Science Open Journal of Cybersecurity

Malware: The Never-Ending Arms Race

Héctor D. Menéndez*

Middlesex University London and Endless Science

h.menendez@mdx.ac.uk

Abstract

"Antivirus is death" and probably every detection system that focuses on a single strategy for indicators of compromise. This famous quote that Brian Dye –Symantec's senior vice president– stated in 2014 is the best representation of the current situation with malware detection and mitigation. Concealment strategies evolved significantly during the last years, not just like the classical ones based on polymorphic and metamorphic methodologies, which killed the signature-based detection that anti-viruses use, but also the capabilities to fileless malware, i.e. malware only resident in volatile memory that makes every disk analysis senseless. This review provides a historical background of different concealment strategies introduced to protect malicious –and not necessarily malicious– software from different detection or analysis techniques. It will cover binary, static and dynamic analysis, and also new strategies based on machine learning from both perspectives, the attackers and the defenders.

Keywords: Malware; Static Analysis; Dynamic Analysis; Concealment Strategies; Machine Learning

1 Introduction

Once upon a time, in 1944 a bug –a real bug– entered into the bulbs of a massive computer disrupting its functionality for days. The software developer Grace Murray Hopper identified the problem and name what we know today as bugs. This was, indeed, the first step to what it would become today's malware. Some years later, in 1971, Bob Thomas, at BBN Technologies, developed the first artificial virus, called the Creeper system, as a proof of concept [65]. They wanted to prove that it was possible to create a

*Corresponding Author

malicious piece of software, based on John von Neumann's "Theory of self-reproducing automata" [76], and indeed it was.

The following years started with the first malware attacks, always attacking vulnerabilities related to human weaknesses. Some examples are: the Jerusalem Virus or Friday 13th (which removed every file in a system every Friday 13th) [16]; and the ILOVEYOU worm [51], a phishing attack that used social engineering to infect multiple machines through a malicious email. Some of this malware enters the system using human intervention or infecting USBs or floppy disks[1]. In this period malware used to be very disruptive, and it was relatively easy to identify it.

It was years later when it started to become a weapon used for and against governments, and indeed against civilians. There are several cases, one of the most popular is Stuxnet in 2010 [55], a malware created probably by some government[2] that infected every possible Windows device until it reached the centrifugate refrigerators of the Iranian nuclear plants, disrupting their plans of nuclear development by spoiling their refrigeration systems. Another interesting example is WannaCry in 2017 [73], a ransomware that encrypted all the personal files in several users' machines, asking them for a ransom to recover their information. Ransomware became very popular after this attack.

Detecting malware is a very difficult task. Consider the scenario: you have an invisible enemy that uses resources that are sometimes wider and better than yours, this enemy will attack from nowhere and you need to be ready for every possible attack that they are going to perform. Indeed, they say that it is easier attacking than defending, but in this arms race, malware is light years beyond defenders. For every step we do, the attackers find a countermeasure, but we have still found responses to multiple attacks that have been disrupting the peace of software during the last 50 years.

When we think about detecting malware, we normally start turning to anti-viruses. They are the first line of defence, and therefore they are well-known by the malicious attackers. Anti-viruses normally perform the first step of malware analysis, called binary analysis. This step tries to identify any kind of signature that can target the software as malware or benign-ware [75]. When this step is not enough, analysts tend to use static analysis, which analyse the software without running it [74]. This helps to identify potential indicators of compromise that might indicate whether the software is malicious or not. If this is not enough, the next step in the triage is dynamic analysis, meaning: run the software and analyse what is happening. This step is time-consuming, but, at the same time, it can be extremely informative, because it provides accurate information about the malicious behaviour [105]. Last but not least, if nothing works from the previous triage steps, you need to reverse engineer the software, and that requires a strong manual effort from a professional analyst to determine whether the sample is malicious or not. Normally this process gives you all the information about the piece of software if it is successful [114].

Several of these steps have helped analysts to identify malware in multiple different contexts, and also some zero-day threats [37], meaning threats that have no been seeing before. However, as I will discuss in this humble review, malware grows and evolves, developing multiple concealment strategies from binary-based detection (see Section 3), static (see Section 4) and dynamic analysis (see Section 5). But even with these techniques affected by the concealment strategies, the new promises of machine learning have been filling for years the security conferences and journals (see Section 6).

[1]If you are 20 or younger, the floppy disk had the shape of the "save file" icon.

[2]Probably a very powerful Western government in tandem with a Mediterranean ally.

Is machine learning promising? Yes. *Is it strong?* Yes. *Is it infallible?* Nope. Machine learning was another step in the arms race, but it has limitations and the adversaries discovered and exploited them [20]. Even more, machine learning can be used to create more sophisticated kinds of malware that learn to conceal from specific technologies [72]. As I said, this is an arms race, every step will have a countermeasure and every countermeasure might be either incremental or disruptive.

This short review about my experience in the malware world is part of my dedication to understanding the different dimensions of this massive problem. Several parts come from my experience at UCL when I was working with Dr David Clark and Professor Earl Barr on malware detection and evasion, others come from my multiple collaborations and discussions with Dr Guillermo Suárez-Tangil and others come from my recent experience acquire in memory analysis, that I learnt during my period at Middlesex. This does not pretend to be a survey but a gentle introductory walk through the malware world, that I have done during these years with these amazing collaborators. In the end, I also leave a short list of tools and datasets that I either used or recommend for analysing malware following any of the previously mentioned strategies (see Section 7).

2 Background

The definition of malware is discussed by several authors [53]. There is no consensus for a final definition, but there are several features that lead analysts to classify software as malicious. These features are related to what the software aims to do:

- it performs a harmful action that affects the user's machine or data, for example, ransomware;
- it steals resources from the machine to perform a specific task, where the software owner will get a specific benefit, for example, crypto-miners;
- it steals user's data or changes the advertisement of an application, for example, spyware or adware;
- it opens a backdoor or reduces the security of either the computer or the network, typically something that a trojan horse or a worm might do.

These features are behaviours and can not be described without "understanding" them, meaning that there is a semantic component that machines are not yet ready to discern, although there are several trails in this direction [84]. But this does not mean that the game is over, it is the beginning. Even if these features can not be semantically understood, they leave "clues" that we can identify, called indicators of compromised [77].

The first step in the arms race was to define these indicators of compromise, and they were defined in the form of signatures [23]. If a piece of software has a specific signature that belongs to malicious software, that software is automatically classified as malicious. This has been the main anti-virus logic since the beginning. These signatures started as hexadecimal codes but these days they are more sophisticated. The Yara project [3] is a good example of the level of sophistication that the signatures can reach [18]. Yara is one of the most popular open-source signature-based detection tools, and it is also associated with the Yara-project[4] that provides multiple signatures for malware detection, among others.

[3]`https://yara.readthedocs.io`

[4]`https://yara-rules.github.io/blog/`

But, as I said, this was the first step of an arms race, and it generated a response. If the anti-virus can detect the malware using a piece of it, the malware only needs to change that piece or conceal it [93]. For that, two main moves responded to signature-based detection: polymorphism and metamorphism. Both methodologies reshape the binary files to make them undetectable under these premises [60]. Polymorphism encrypts the binary and adds a decryption stub –or code– that differ from a malware variant to another. Metamorphism uses equivalent instructions to replace the original ones and recompiles the code when needed.

This was a game-changer because the anti[viruses passed from detecting specific malware to detect specific variants and, at the same time, these variants grew exponentially. Therefore, the focus changed the detection of signatures to traces or dynamic signatures. However dynamic signatures forced the analyst to run the malware and that reduced the analysis scalability [81]. And even if these signatures avoided the polymorphic and metamorphic encryption by paying the scalability price, the malware developers found a way to avoid the dynamic-based detection, by including mechanisms that would change the behaviour of malware depending on the circumstances [108]. Some good examples are those concealment strategies focused on detecting virtual machine-based environments [45] or the debugging mode [17]. Other examples targeted specific devices or specific users and do not react until they reach their target [19]. This was again another disruptive move against malware mitigation, but it was not the end, as analysts always aimed to find new techniques to detect malware.

During the following sections, I will define these concepts and continue with the story of the arms race, trying to cover the main areas of the concealment strategies and keeping a historical perspective of the different steps.

2.1 Types of Malware

Malware has many faces and, as the definition itself, it is general enough to provide strong headaches to analysts and researchers on the topic. Besides, it evolves and changes, therefore, new kinds of malware will appear in the following years. Just to mention some of the classic and actual types of malware, we have:

- **Viruses**: this is the most classical kind of malware and it is normally defined as the biological virus. A malicious piece of software reproduces itself infecting every machine that it touches and propagates through other machines, normally via a USB stick or infected device [82].
- **Worms**: similar to viruses, worms propagate through the network and infect every machine connected that is vulnerable to the infection itself [91].
- **Trojan horses**: these types of malware conceals behind a genuine behaviour. The malware performs an expected task, while it performs a malicious activity in the background, for instance, it steals the user's passwords [49].
- **Ransomware**: this type became more popular a few years ago with the attack of WannaCry [73]. This type of malware normally encrypts all the files in a system and asks the user for a ransom to obtain the decryption key and recover the file system.
- **Rootkits**: this type of malware normally works in kernel or admin mode. Exploiting some vulnerabilities, it tends to create a new kernel module in the system to obtain root (or admin) privileges and take control of the whole machine without the user's consent [26].

- **Spyware**: this malware is normally a kind of rootkit that activates some of the user's devices, such as the camera or microphone, to obtain the user's information in real-time [46]. This is normally used to spy on politicians, journalists, or activists, such as the recent Pegasus operation [50].
- **Adware**: this malware aims to modify the advertisement of the different system apps to obtain the advertisement payment that would correspond to a genuine app. Normally this kind of malware is either a trojan horse, such as a browser add-on or an app that is cheaper than the original (but it is just the original repacked) [58]. It is normally detected because of the advertisement cookies, that do not corresponds with the user's preferences [44].
- **Botnets**: this malware infects machines and waits for orders from a command and controller machine [34]. This is one of the classic methodologies to create computers networks for performing a specific task. The infected machines connect with one or many controller machines. These controllers coordinate the communications between the attacker and the infected machines. When the attacker requires a strong computational power to, for example, attack a strong network of servers, it coordinates all the infected machines to perform a denial of services attack or any other kind. An example was the Mirai botnet coordinated attack using IoT in 2016, which infected multiple smart TVs to perform a denial of service to different popular services such as Netflix, Twitter and Airbnb, among others [4].
- **Crypto-miners**: This kind of malware aims to conceal from the user as much as possible and it uses the computer resources to mine crypto-currencies. This malware has generated a strong economic impact in the dark web market as Pastrana and Suárez-Tangil studied by extracting transaction information from crypto-miner's pools [80].
- **Autoimmune**: This malware cheaps the system to attack itself. For example, a benign app with a malicious signature would be the target for an anti-virus. Imagine that multiple variants of a specific malware family constantly repeat a signature, and that signature is a popular app (for instance, a Google product), then the anti-virus will learn the signature and mark that benign app as malicious, removing it from the system [72]. This happened to me when I was studying the response of different anti-viruses to an intelligent packer. The moment we added benign-ware to the packing process the anti-viruses started to generate false positives [72].

Although these are the most common kinds of malware, there are other types that are either new or variations of the previous ones. It is also possible that, at some point, the extension of malware will bring a new dimension when we will reach the limits of biology. If the malware can ransom, for example, a bypass or an implant, it would become a potential digital-to-biological virus.

3 Classic Concealment: Poly and Metamorphic

Classic concealment aims to cover binary signatures. To reach that goal, the binary files need a different shape to the original file where the signature was extracted [85]. With that purpose in mind, the attackers created methods that reshape the binary files based on encryption and compression. These methods are called packing, polymorphism and metamorphism [21].

3.1 Polymorphism and Metamorphism

Packing does not modify the semantics of the program, it only modifies the way it is stored. Given a program binary, such as a PE (Program Executable, normally on Windows systems) or an ELF (Executable Linkable Format, normally on Unix/Linux) file, we can apply a packer and create the pack version of this program. The binary file is divided into sections, for instance `.data`, `.text`, `.rscr`, `.bss` (see Table 1). These sections normally correspond with different program parts. The packer will modify these sections and create new ones. For example, a popular packer is UPX [96]. UPX takes all the sections but those related to the imports (neither the overlay in the case of PE32) and compresses them into a single section called UPX1. The import names are also taken and stored in a section called UPX2, as a list of names, so the executable file is no longer linked to these libraries. UPX adds a code, called stub, in the UPX1 section that will decompress the original section in execution time and will reconstruct the table of imports by reading the UPX2 section. This means that the stub will be the entry point of the new version of the program. For the import reconstruction, UPX uses four functions that will read each function name and link it with the proper library in execution time.

This process allows to significantly reduce the file size and, considering that UPX is not malicious per se, it also conceals any signature of the original binary, considering that the packed file will be a different binary file after the compression. However, this only generates a variant of the malware. To extend this to multiple variants, it is possible to add a modification to UPX that also adds XOR encryption to the compressed area, keeping the key in the stub [71]. This encryption gives you up to 255 more variants if the key is one byte long, or more depending on the encryption strategy. This strategy led to oligomorphic engines, that allow to conceal the malware and add multiple different types of stubs giving millions of potential variants [72]. The stub is normally modified to make sure it does not become the binary's signature. Oligomorphism has a set of stubs to choose from, but at some point, anti-viruses would be able to have the whole list. For that reason, malware engineers created polymorphism, which allows creating as many stubs as they need to conceal their malware and make it completely undetectable by using metamorphic variations of the stub [21].

Although polymorphism is a popular method to conceal malware, it is sensitive to memory analysis. When the program runs, it goes into memory. Then, the stub

Section	Description
`.data`	Program's data.
`.text`	The program's code.
`.rscr`	List of imports.
`.src`	Extra imports and libraries.
`.bss`	Uninitialized variables.
`overlay`	Extra information that will be read as a file (win32 only).
`UPX0`	Empty section to decompress the original code.
`UPX1`	Compressed code and execution stub.
`UPX2`	List of imports and overlay.

Table 1: Sections of a PE/ELF binary file in the system after linking either before (top) or after packing (bottom).

will decrypt, decompress and run the original code. Either the whole program will be decompressed and decrypted, as is the case of UPX, or parts of the program, as is the case for Themida [95]. In both cases, the original code –or part of it– will be exposed. Having a system that can monitor the RAM, such as an online version of Volatility [62], the signature can be detected. Another popular method to detect polymorphic malware focuses on detecting the polymorphic engine [3]. Polymorphism leaves a signature in terms of entropy. Compression and encryption increase the entropy of a binary file, because of the aim to condense the information and make the file looks as "random" as possible. This helps to know whether a binary is using these kinds of protection, and it can be considered as an indicator of compromise. However, the entropy of a file after packing can be controlled, as my colleges and I proved with our tool EEE (The Evolutionary Packer) [71], which introduced controlled entropy regions in packed files.

Another popular strategy for malware concealment is metamorphism [113]. Similarly to polymorphism, metamorphism modifies the binary file, but at code level (normally a disassembled version of the code), creating variations of the code instructions. These variations break signatures, and they are diverse enough to become hard to detect by anti-virus engines. Moreover, once the executable file is in memory, the signature can not be detected as it was the case of polymorphism because the new code is semantically equivalent and therefore it does not need any runtime modification to run. This makes it harder to detect metamorphic engines. Nonetheless, the metamorphic engine can also leave signatures that can be detected by static analysers if the file has no extra protection. Examples go from the strings to specific functions calls or flows within the program. It is important to include other strategies to conceal the files, even if they are already strong in front of anti-virus based detection.

3.2 Detecting Polymorphism and Metamorphism at Binary Level

When analysts aimed to face the challenge of detecting polymorphism and metamorphism, they needed to perform the first disruptive move in the arms race. Looking for options moved them to strategies based on information theory, mainly n-gram and entropy-based detection methods.

Some works that started with this kind of detection was mainly based on the average entropy of a file. The work introduced by Lyda and Hamrock [64] in 2007 was able to detect the entropy signature of the polymorphic engine. Then, the work of Li et al, based on n-grams, tried to create a reconstruction of the malicious file in terms of byte sequences [59]. These sequences showed the most common file patterns used to detect whether it was malicious or not. Several of these techniques were currently combined with machine learning (see Section 6).

The main idea behind binary-based detection systems was to create a set of features that can provide enough information to the machine learning classifiers to distinguish malware or benign-ware. And this was indeed a successful step, at least for a while. Some good examples were the work of Mark Stamp and colleges on structural entropy [11], which extended the work of Ivan Sorokin by creating entropy profiles [97]. Sorokin's technique created a profile per binary file with a specific signature size. With this profile, they were able to create file segments and compare different files based on these segments. Stamp's team extended this feature space with machine learning. Even though this idea was good, it missed information on the profile and its scalability was quadratic. In that period, we consider improving the scalability of this technique using time series analysis, and we created the entropy time series (or EnTS) [12], that obtained better results and scale linearly, which means that it was about 1000 faster than other techniques of the state-of-the-art, for instance, another technique called normalised compressed distance, which was better in terms of detection than ours but 3000 times slower [3]. However, this part of the story ended in a bitter-sweet way. We are able to detect malware variants, but we generate false positives, which is dangerous for anti-virus systems, due to benign applications will be targeted as malicious. Other ways of analysis are needed and for that reason, we kept exploring static and dynamic analysis.

4 Anti-Static Concealment

Binary-based detection is not enough, especially when the attackers use a strong concealment strategy. Besides, when the attackers create new malware, they can also recycle old indicators of compromise that can not be discovered by these techniques. Then, the next step of the triage is to understand the malware from a static analysis perspective, where the malware is analysed but not executed [25].

Static analysis was introduced to understand the different behaviours of malware, for instance, the code [88], its system dependencies [104], flows between inputs and outputs [6], third-party libraries used [68], permissions [7], and static behaviours [2], among others.

There are multiple examples of these specific features that are indicators of compromise. The most classical ones are on Windows, for instance, Santos et al. [88] analyse opcode from reversed malware in order to identify malicious sequences. During my work on this topic, my colleges and I [68] focused on the analysis of API calls also aiming to detect those sequences of API calls that might be either scaling privileges or attacking the system. Similarly, we can find third party calls as one of the main detection strategies for two reasons: 1) the combination of sequences of these calls might lead to malicious behaviours, and, 2) they can not be obfuscated within the original malware [2].

Focusing on Android malware analysis, we see more sophisticated techniques. A good example is FlowDroid [6]. This tool identifies paths within the code between a specific source of data to a specific sink –for example, a password as a source and an SMS as the sink. When it finds a flow between the source and sink, it sets a flag on that flow. Another example is related to the permissions of an Android app [7]. Permissions have proven to be one of the most accurate methodologies to identify malware for a long time, mainly because some malware needs specific permissions to operate. However, this changed during the last years and malware started to focused on leveraging other apps to attack the systems, via intent actions [35], i.e. public interfaces that some apps can leverage from others, such as the photo camera.

Other relevant static strategies are taint-analysis, which aims to follow variables within the program code or API callbacks [39]; symbolic execution to identify, for example, potential payload triggers within the malware by finding the input or the environment that would activate a suspicious piece of code [110].

4.1 Obfuscation Methods

Although static analysis is a powerful tool to detect malware, it relies a lot on access to the source code, which good attackers conceal not just with polymorphism and metamorphism but also with other static analysis techniques that try to reduce the possibility of detecting indicators of compromise.

The first obvious technique, which would be close to metamorphism, although the latter applies to binary level, is code obfuscation [24]. Code obfuscation has several levels of sophistication and has been used to conceal the code behaviour by making it unreadable. Some good examples of obfuscation strategies are renaming variables or changing arithmetic operations [32]. However, there are others more sophisticated. Collberg et al. [24] describe several of them their taxonomy of obfuscation. The following lists some obfuscations that can be found in Tigress [10] or other tools and that affect static analysis:

- Breaking methods: this obfuscation takes a method and divides it into two or more methods to make the analysis harder.
- Anti-Taint analysis: this obfuscation generates noise during the taint analysis process by adding variables or other elements to the code. This makes it hard to follow a specific static trace.
- Virtualization: this obfuscation creates its own interpreter on top of the original program and rewrites the program in its own language, making any reverse engineering or disassemble process extremely hard.
- Opaque Predicates: this obfuscation adds predicates or branches to the program that will never be traversed, therefore, the code within these program parts will not be executed. The conditions for these predicates are not easy to understand and the static analysers normally assume that they will be executed.
- Hashing or Encrypted Strings: it is also common to conceal the strings as they can be indicators of compromise (for example a URL or a Bitcoin wallet).
- Reflective programming: another obfuscation strategy is to leave some code as a string (normally encrypted), and load it in runtime (similarly to packers). This strategy significantly affects several static analysis tools that tend to ignore these parts of the program.

Although all these techniques are focused on facing static analysis, several of them can not deal with dynamic analysis, for that reason, when a piece of software is difficult to understand from a static perspective during the triage process, it is normally submitted to the dynamic analysis team.

5 Anti-Dynamic Concealment

Dynamic analysis is the next step during the triage process, where the analyst needs to run the malware to understand its behaviour. The analyst uses a virtual machine or sandbox environment to perform this kind of analysis, keeping information of the different malware traces produced, for instance, network packets, files created, URLs connections, flows between different elements of the binary file, among others [78].

Concealing from dynamic analysis is a compromise between altering the malware behaviour when the malware is within the sandbox and performing its normal operations when it is within the infected system. This leads the malware to activate specific behaviours under specific contexts. These activations that alter the malicious behaviour are either call triggers when the malware release a payload [19] or red-pills when the malware detects the virtual environment [79].

5.1 Concealing from the Sandbox

There are multiple techniques that malware uses to conceal itself from the sandbox. Considering that dynamic analysis has a strong time component, the malware needs to wait long enough to make sure it does not release any indicator of compromise. Some of the most relevant techniques are:

- **Packing Memory Protection**: several malicious software employs packing to prevent static analysis (see Section 4). Dynamic analysis was presented as a potential solution, considering that the software needs to unpack itself in memory before running. However, the levels of sophistication of some packers like Themida [47], also keep this into consideration during the development process. To make sure that not all the code is visible even in run memory –when the malware is running–, the packer performs its task by blocks, encoding the blocks that are not used once they are executed and decoding them only to run.
- **Metamorphism and Obfuscation**: These two techniques are also present in static analysis and also prevent any kind of dynamic analysis considering that even when the code is visible, the analyst has trouble understanding it [9].
- **Anti-virtual machine**: these methods use the so call red-pills to detect the environment. When the malware detects a virtual machine it can decide to stop running or delay some instructions based on the virtualization environment. Sometimes, when the malware is already compromised, it conceals part of its behaviours so only known indicators of compromise are release and others that could identify variants keep hidden [45]. This prevents the malware to scale to the reverse engineering step of the triage process.
- **Anti-Debugging**: some analysts use debugging as a controlled methodology to analyse malware and skip some code sections to focus on indicators of compromise. Even though this process is very close to reverse engineering, there are also automatic debugging processes that can carry out a malware analysis [8]. However, debugging also requires activating some specific flags during the process execution that the malware can detect to conceal some parts of its code and make the analysis harder [22].

These are just a few examples of how the malware can detect the virtualization or analysis environment to conceal its behaviour. The following shows specialized design malware that can only be detected using dynamic analysis.

5.2 Concealing from the Analyst

It is important to recall that dynamic analysis can detect some kinds of malware that can hardly be detected by static analysis and, at the same time, it is a strong tool to detect indicators of compromise. These kinds of malware are:

- **Fileless Malware**: This kind of malware leaves no trace in the hard drive, keeping itself as part of the RAM in an active process. It can perform operations for persistence like modifying the Windows registry or creating specific services that download or generate the malware once the operating system starts [54]. The only way to detect this kind of malware is to perform a dynamic analysis of the principal memory (or RAM) and identify the persistence strategy and the concealment method.

- **Return Oriented Programming**: This kind of malware leverages a combination of the stack and other processes code to generate its malicious behaviour by reading the code and combining return operations [83]. The main component of this malware is its lack of action *per se*. It does not have any malicious code but a combination of actions that activates malicious codes in others.
- **Hijacking**: If the malware can not conceal itself within the system, it can hijack a specific function from a system's library [87]. This method consists of replacing the legitimate function with a malicious one. It satisfies two actions, 1) it guarantees that the malware is not removed from the system when the user tries to remove it, and, 2) it runs the malicious payload. For example, if the malware wants to read any buffer related to the user's information, it can hijack the memory functions that control the heaps [62].

Normally these cases are identified through their persistence mechanism and it requires memory analysis to be able to acquire enough information to study them. An interesting and powerful tool for this is Volatility [62] that provides several different components to analyse RAM snapshots for Windows, Linux and Mac.

6 Learning-based Detection

The previous sections focused on different perspectives for malware analysis. In these cases, the analyst aims to understand how malware works and what can be done to detect indicators of compromise. However, sometimes these indicators of compromise are hidden, and identifying malware requires identifying other kinds of patterns at a statistical level. Other times, it is tedious to apply the same analysis several times, and we need a way to identify similar data related to malicious software to accelerate the detection process. This is how machine learning is applied to malware detection to discriminate malware from benign-ware [99] or identifying malware families [20].

This process uses a methodology that extracts features from the software under analysis. These features can either be from the binary file itself (for example, the entropy [64] or the n-gram distribution [59]), from static analysis (for example, opcodes from the disassembly version [88] or flows from the program [6]), and from dynamic analysis (network traces [81] or files created [103]).

Once we have the process to extract the features, we select a corpus. Depending on the goal, either malware detection or malware families classification, we choose a corpus of malware and benign-ware, or malware families. It is important to guarantee that every targeted class, meaning the two classes of malware/benign or the n-classes related to the n-families you need to classify, are balanced [56].

Once the data is ready and analysed, we can train a machine learning algorithm for creating a model [56]. The model will learn from the features how to discriminate the classes, for example, malware and benign-ware. Every new sample that we pass through the model (or test) will be classified according to this learning process in the most probable class. The advantage of these models is the ability to automatically learn from the data and to extrapolate their knowledge in a way the can seriously accelerate the malware detection and classification process. This led to a significant series of contributions from multiple perspectives of cybersecurity that relayed in machine learning.

There are multiple contributions to the analysis of malware based on using binary, static and/or dynamic features to feed machine learning to detect malware. Some examples are already mentioned, such as structural entropy [97], and its extensions to metamorphic malware detection [11], time-series features [71], and mimicking antivirus behaviours [70]. Other good examples –more classical–, are the work Schultz et al. [92] who used n-gram features to feed different machine learning approaches for Windows malware detection; Kolter and Maloof who also [52] included the information gain metric and byte-level analysis to the n-gram features; Stolfo et al. [98] who applied a similar methodology to PDF malware detection; Tabish et al. [101] who combined the n-grams with histograms to extract different metrics from information-theory; and Santos et al. [89] who used n-gram features but reduced the labelling process by using semi-supervised methods. The work of Alshahwan et al. [3] is one of the last applications of these techniques. In this work, we compared different information-theoretic techniques combined with machine learning to detect Windows malware using different concealment strategies. Other modern examples have been applied to Android malware detection, for example, the work of Martin et al. [67] combining n-grams and Android permissions to detect Android malware.

Static analysis is significantly more popular in terms of identifying features for malware. Examples are using the control flow graph [90], op-code analysis [88], PE headers and API calls [111]. However, as Moser et al. highlighted [74], it has significant difficulties to deal with concealment strategies. This is partially because these techniques tend to rely on taint analysis [36] or symbolic execution [86]. They are more popular for Android systems, where the concealment strategies are not as strong as for Windows systems. For example, CHABADA [42] uses API calls from the apps combines with unsupervised learning, MOCDroid [68] combines them with genetic algorithms, and the work of Yerime et al.[112] combines them with Bayesian classifiers. Another popular methodology is RevealDroid [38] that combines multiple static features such as permissions, API calls, intent actions and flows. In this part, we could also consider tools like ADROIT [66], which uses the meta-information of the malware to detect it.

In terms of dynamic analysis, multiple machine learning applications combine information extracted either from the malware execution in virtual machines [30, 106] (even though red pills can alter it, as I mentioned in Section 5), or network traces [81]. Some interesting applications focused on Android are Drebin [5] and CopperDroid [102] which create behavioural models using the execution traces, and apply anomaly detection to detect malware. Another example of these tools is ALTERDROID [100], which can detect obfuscated components from behavioural traces. There are also some cases where both static and dynamic features are combined. Tools like AndroPyTool have been designed for this purpose and a popular dataset created with this tool for machine learning applications is the OmniDroid dataset [69].

To guarantee that machine learning has enough data to be applied, it is important to identify the proper datasets. Section 7 provides different datasets that can be used for malware analysis.

6.1 Adversarial Machine Learning

Machine learning looked like the holy grail in the malware arms race. Unfortunately, this golden grail was not the right one. Machine learning parts from the assumption that the test and train sets follow the same distribution. This statistical assumption is useful for those cases where the data has no intelligence behind it. But in the malware arms race, we always have to consider that our adversary or attacker will perform a step forward the moment that we figure out a defence. This step forward, in the machine learning context, is called adversarial machine learning [13].

Adversarial machine learning studies the limits of machine learning systems. These might be affected by the boundaries between classes, data bias, wrong selection of parameters, over-fitting, or wrong selection of the classifier. The main idea of the application of adversarial machine learning to malware detection is to be able to predict which incremental steps an adversary can perform to find a misclasification. This idea, which my colleges and I named *hothousing* [72], will force the adversary to perform a disruptive move in the arms race to prevent the detection, as every incremental movement is supposed to be contemplated. Some authors consider this scenario as a zero-sum Stackelberg game [117], where the defender has to wait for the next step of the adversary to respond.

Depending on the adversary's knowledge, there are four levels where the adversary can work [15, 57]. The highest (level 3) assumes that the adversaries have knowledge about the data, features and classifier used, and they have direct access to the classifier or can create an equivalent one. The next level (level 2) assumes that the adversaries have access to the classifier and they know which features are used during the training process. Level 1 only allows information about the classifier, while level 0 only provides access to the final decision (malware or benign-are).

In the malware context, adversarial machine learning has been applied to multiple problems. One of the most popular is evadeML [109] which is a genetic-programming system (in level 3, meaning with all knowledge) that manipulates PDF malware to create variants that can evade detection. The success of this methodology led to multiple works in the area. Some examples where I was involved are IagoDroid [20], which is also a level 3 system that uses evolutionary computation to evade family classification. It proved to defeat RevealDroid with no effort. Another similar methodology applied to Windows malware is EEE, the evolutionary packer. This packer is the only one in the literature that proves its effectiveness as a level 0 system, defeating multiple anti-viruses [72] and machine learning algorithms [71], by controlling the entropy of malware variants.

Adversarial machine learning can also been seeing depending on the machine learning part that the attackers target. For example, Globerson and Roweis [40] focused their attacks on manipulating sample features aiming to evade a support vector machine, Zhou et al. [116] defined an attack directly on the support vector machine classifier, similarly to EvadeML [109]. Other interesting works in this area are the works of Battista Biggio. Some examples can be found on the compilation of Biggio and Roli about applications of adversarial machine learning [13]. There are also interesting examples of what adversarial machine learning can reach. Some are focused on data poisoning [14] or even attacking the malware triage [20]. There are also some cases studying the transferability of adversarial samples [28] where a sample that successfully evades a classifier can evade a similar one having no knowledge about it.

There are also works where the defender aims to adapt to the attacker. For example, Kantarcioglu et al. [48] modified the classifier's cut-off to be more resistant to the evasion attack, while Lui and Chawla [63] use a game theory approach based on identifying the Nash equilibrium between the attacker and defender. Another interesting case is the study between EEE (the evolutionary packer) and VirusTotal, where VirusTotal learns from EEE's attacks [72].

Even though this looks like the end of this story, malware keeps evolving and defenders keep developing new techniques to detect them. The next section will provide some tools and data that can be useful for anyone who wants to start in this field.

7 Malware Analysis Tools and Dataset

In order to start with malware analysis, there are two main things that everyone needs: tools and data. As a researcher, you will need to obtain datasets with real malware or malware information and tools that allow you to manage that data or analyse it.

7.1 Tools

In order to obtain a ground truth about the software, meaning either whether it is malware or specific malware families, one of the first tools that any researcher needs is VirusTotal[5]. Virus total has a corpus of anti-virus (around 80 at the moment) analysing the malware and providing a ground truth and a label for the malware. For malware families, it is likely that the anti-viruses disagree, therefore you need to find a consensus among them. AVClass [94] helps to determine the best family classification. It is also important to consider that some anti-viruses will be predictable. MimickAV [70] will allow determining which anti-viruses are more predictable and therefore vulnerable to potential evasion attacks based on the transferability of evasive samples [28].

For binary analysis, there are some tools that help to apply signatures either directly to the malware samples, like Yara[6]. Another option to extract entropy profiles is EnTS[7]. For reverse engineering, there are several tools, the most relevant are Radare2[8], Ghidra[9] IDA Pro[10], the GNU Debugger (GDB)[11], OllyDbg[12], the disassembler Capstone[13], and in the case of Android code, Androguard[14].

The case of static analysis is more diverse. The tools that have proven to be more effective are related to Android analysis. Good tools to perform static analysis are FlowDroid [6] and DroidSafe [41], which analyse whether there is a flow between a specific source of code and sink. If you are creating a static analyser, you will need to compare it with some state of the art systems. These are, for instance, Kirim [33] and DroidRanger [61] which focuses on detecting malicious permission setup; DroidAPIMiner [1] and DroidLegacy [29] perform a similar task by analysing API calls, and RevealDroid [38] which aggregate with information with flows and intent actions.

For dynamic analysis, we can identify more general tools. For instance, Volatility [62], the memory forensic tool, that can identify malware traces in RAM memory after the malware runs; strace [15] and Triton[16] that analyse malware traces; Cuckoo [78] and PANDA [31], which are virtual machine environments that run malware and extract all the different dynamic traces created during the execution; and Wireshark[17] a tool that analyses the network traces. For specific applications to Android, there are tools like CopperDroid [102] for general dynamic analysis; IntelliDroid [107] to find malware

[5]https://virustotal.com
[6]http://yara.readthedocs.org and http://yararules.com/
[7]https://github.com/hdg7/EnTS/
[8]https://radare.org/
[9]https://github.com/NationalSecurityAgency/ghidra
[10]https://hex-rays.com/ida-pro/
[11]https://www.gnu.org/software/gdb/
[12]http://www.ollydbg.de/
[13]https://github.com/aquynh/capstone
[14]https://github.com/androguard/androguard
[15]https://sourceforge.net/projects/strace/
[16]https://triton.quarkslab.com/
[17]https://www.wireshark.org/

triggers; and VizMal [27] for visualizing the analysis results.

There are several more tools for malware analysis, if the reader wants to refer to a more extensive list of tools, I strongly recommend having a look at the Awesome Malware Analysis List[18].

7.2 Malware Data

There are several sources of malware data. Some of them are:

- **VirusTotal**: Apart from being a strong tool for analysing malware, VirusTotal also provides malware data, especially under the academic license. These malware samples are already analysed and the tool provides significant information related to its behaviour.
- **VirusShare**[19]: VirusShare provides lived malware and is updated about every month. It provides any kind of malware sample, for instance, Windows, Linux, Android, Mac, PDF, Doc, and JavaScript, among others.
- **Drebin**[20]: This classical dataset is used for Android malware families. It has 5560 samples related to 179 Android families. Although this dataset has been widely used in Android malware analysis, it is still very popular and can be found in multiple research papers.
- **Kaggle Malware Dataset**[21]: This dataset was released in 2014 for a Kaggle competition. This dataset contains 10,869 malware files in its training set, belonging to 9 different families. It is used in research for experiments for family classification and malware detection. The data has the disassemble code and the byte-code.
- **Genome**[22]: The Android Malware Genome Project [115] has also a classic dataset of malicious Android apps, which has widely been used in the literature (it is important to recall that this project has been abandoned in 2015). The dataset contains 1,260 malware samples of 49 different families extracted between 2010 and 2011 from Android markets.
- **AndroidZoo**[23]: this dataset is becoming one of the largest Android malware datasets with 16,355,705 Android apps analysed by different anti-viruses. The malware labels are obtained via VirusTotal and Euphony [43], a system that infers the final classification.

There are also several services that provide information about malware samples. It is important to choose the data according to the main goals for the analysis.

8 Conclusions

One step after another the malware arms race creates more sophisticated software and security protections, even if the cost is high in terms of the current software resilience.

[18]`https://github.com/rshipp/awesome-malware-analysis`
[19]`http://virusshare.com`
[20]`https://www.sec.cs.tu-bs.de/~danarp/drebin/`
[21]`https://www.kaggle.com/c/malware-classification`
[22]`http://www.malgenomeproject.org/`
[23]`https://androzoo.uni.lu/`

Indeed the attackers contribute to creating troubles for governments and companies, but defenders are reaching strong levels of sophistication in their efforts to stop them. Semantic analysis, symbolic execution, memory forensics, flow analysis, reverse engineering tools and dynamic analysis tools are some good examples of all the good software that this arms race is bringing to our side.

It is also important to remark that the defenders' culture is based on sharing information. Protecting us from malware threats requires sharing and, therefore, multiple tools that we are using these days for malware analysis either have an open or a free software license. We know that we will not be able to stop every possible attack, but during the learning process, we will reach new knowledge that will also help to solve other problems in the software world.

Acknowledgements

Formally, this paper has been developed as part of the MLighter (or AdvMLTester) project funded by Innovation UK. However, the main contributors to all this knowledge are my two mentors at UCL: Dr. David Clark and Professor Earl T. Barr; my main collaborator in malware analysis, whom I teach about malware mitigation at Oxford University, Guillermo Suárez-Tangil and all the collaborators that I had during ths process that started in April 2015, when I joined UCL to work in malware analysis.

License

References

[1] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013. doi: 10.1007/978-3-319-04283-1_6. URL `http://dx.doi.org/10.1007/978-3-319-04283-1_6`.

[2] Mamoun Alazab, Sitalakshmi Venkataraman, and Paul Watters. Towards understanding malware behaviour by the extraction of api calls. In *2010 second cybercrime and trustworthy computing workshop*, pages 52–59. IEEE, 2010. doi: 10.1109/ctc.2010.8. URL `http://dx.doi.org/10.1109/ctc.2010.8`.

[3] Nadia Alshahwan, Earl T Barr, David Clark, George Danezis, and Héctor D Menéndez. Detecting malware with information complexity. *Entropy*, 22(5):575, 2020. doi: 10.3390/e22050575. URL `http://dx.doi.org/10.3390/e22050575`.

[4] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

[5] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[7] AM Aswini and P Vinod. Droid permission miner: Mining prominent permissions for android malware analysis. In *The Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2014)*, pages 81–86. IEEE, 2014. doi: 10.1109/icadiwt.2014.6814679. URL `http://dx.doi.org/10.1109/icadiwt.2014.6814679`.

[8] Mikhail Auguston, Clinton Jeffery, and Scott Underwood. A framework for automatic debugging. In *Proceedings 17th IEEE International Conference on Automated Software Engineering,*, pages 217–222. IEEE, 2002. doi: 10.1109/ase.2002.1115015. URL `http://dx.doi.org/10.1109/ase.2002.1115015`.

[9] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Impact of code obfuscation on android malware detection based on static and dynamic analysis. In *ICISSP*, pages 379–385, 2018.

[10] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200, 2016. doi: 10.1145/2991079.2991114. URL `http://dx.doi.org/10.1145/2991079.2991114`.

[11] Donabelle Baysa, Richard M Low, and Mark Stamp. Structural entropy and metamorphic malware. *Journal of computer virology and hacking techniques*, 9 (4):179–192, 2013. doi: 10.31979/etd.6zrt-mb7y. URL `http://dx.doi.org/10.31979/etd.6zrt-mb7y`.

[12] Sukriti Bhattacharya, Héctor D Menéndez, Earl Barr, and David Clark. Itect: Scalable information theoretic similarity for malware detection. *arXiv preprint arXiv:1609.02404*, 2016.

[13] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018. doi: 10.1016/j.patcog.2018.07.023. URL `http://dx.doi.org/10.1016/j.patcog.2018.07.023`.

[14] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389*, 2012.

[15] Battista Biggio, Giorgio Fumera, and Fabio Roli. Security evaluation of pattern classifiers under attack. *IEEE transactions on knowledge and data engineering*, 26(4):984–996, 2013.

[16] Douglas B Bock and John F Schrage. Computer viruses: over 300 threats to microcomputing... and still growing. *Journal of Systems Management*, 44(2):8, 1993.

[17] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat*, 1:1–27, 2012.

[18] Michael Brengel and Christian Rossow. Yarix: Scalable yara-based malware intelligence. In *USENIX Security Symposium*, 2021.

[19] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008. doi: 10.1007/978-0-387-68768-1_4. URL http://dx.doi.org/10.1007/978-0-387-68768-1_4.

[20] Alejandro Calleja, Alejandro Martín, Héctor D Menéndez, Juan Tapiador, and David Clark. Picking on the family: Disrupting android malware triage by forcing misclassification. *Expert Systems with Applications*, 95:113–126, 2018. doi: 10.1016/j.eswa.2017.11.032. URL http://dx.doi.org/10.1016/j.eswa.2017.11.032.

[21] Silvio Cesare, Yang Xiang, and Wanlei Zhou. Malwise—an effective and efficient classification system for packed and polymorphic malware. *IEEE Transactions on Computers*, 62(6):1193–1206, 2012. doi: 10.1109/tc.2012.65. URL http://dx.doi.org/10.1109/tc.2012.65.

[22] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE international conference on dependable systems and networks with FTCS and DCC (DSN)*, pages 177–186. IEEE, 2008. doi: 10.1109/dsn.2008.4630086. URL http://dx.doi.org/10.1109/dsn.2008.4630086.

[23] Mihai Christodorescu and Somesh Jha. Testing malware detectors. *ACM SIGSOFT Software Engineering Notes*, 29(4):34–44, 2004. doi: 10.1145/1007512.1007518. URL http://dx.doi.org/10.1145/1007512.1007518.

[24] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Citeseer, 1997.

[25] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13 (1):1–12, 2017.

[26] Michael Davis, Sean Bodmer, and Aaron LeMasters. *Hacking exposed malware and rootkits*. McGraw-Hill, Inc., 2009.

[27] Andrea De Lorenzo, Fabio Martinelli, Eric Medvet, Francesco Mercaldo, and Antonella Santone. Visualizing the outcome of dynamic analysis of android malware with vizmal. *Journal of Information Security and Applications*, 50:102423,

2020. doi: 10.1016/j.jisa.2019.102423. URL http://dx.doi.org/10.1016/j.jisa.2019.102423.

[28] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 321–338, 2019.

[29] Luke Deshotels, Vivek Notani, and Arun Lakhotia. Droidlegacy: Automated familial classification of android malware. In *Proceedings of ACM SIGPLAN on program protection and reverse engineering workshop 2014*, pages 1–12, 2014.

[30] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, 2008.

[31] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 839–850, 2013.

[32] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In *International conference on security and privacy in communication systems*, pages 172–192. Springer, 2018. doi: 10.1007/978-3-030-01701-9_10. URL http://dx.doi.org/10.1007/978-3-030-01701-9_10.

[33] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245, 2009. doi: 10.1145/1653662.1653691. URL http://dx.doi.org/10.1145/1653662.1653691.

[34] Maryam Feily, Alireza Shahrestani, and Sureswaran Ramadass. A survey of botnet and botnet detection. In *2009 Third International Conference on Emerging Security Information, Systems and Technologies*, pages 268–273. IEEE, 2009.

[35] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, Guillermo Suarez-Tangil, and Steven Furnell. Androdialysis: Analysis of android intent effectiveness in malware detection. *computers & security*, 65:121–134, 2017. doi: 10.1016/j.cose.2016.11.007. URL http://dx.doi.org/10.1016/j.cose.2016.11.007.

[36] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 576–587, 2014. doi: 10.1145/2635868.2635869. URL http://dx.doi.org/10.1145/2635868.2635869.

[37] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Zero-day malware detection. In *2016 Sixth international symposium on embedded computing and system design (ISED)*, pages 171–175. IEEE, 2016. doi: 10.1109/ised.2016.7977076. URL http://dx.doi.org/10.1109/ised.2016.7977076.

[38] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):1–29, 2018.

[39] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *International Conference on Trust and Trustworthy Computing*, pages 291–307. Springer, 2012. doi: 10.1007/978-3-642-30921-2_17. URL http://dx.doi.org/10.1007/978-3-642-30921-2_17.

[40] Amir Globerson and Sam Roweis. Nightmare at test time: robust learning by feature deletion. In *Proceedings of the 23rd international conference on Machine learning*, pages 353–360, 2006.

[41] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[42] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th international conference on software engineering*, pages 1025–1035, 2014.

[43] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. Euphony: harmonious unification of cacophonous anti-virus vendor labels for android malware. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 425–435. IEEE Press, 2017. doi: 10.1109/msr.2017.57. URL http://dx.doi.org/10.1109/msr.2017.57.

[44] Ianir Ideses and Assaf Neuberger. Adware detection and privacy control in mobile devices. In *2014 IEEE 28th Convention of Electrical & Electronics Engineers in Israel (IEEEI)*, pages 1–5. IEEE, 2014. doi: 10.1109/eeei.2014.7005849. URL http://dx.doi.org/10.1109/eeei.2014.7005849.

[45] Anoirel Issa. Anti-virtual machines and emulations. *Journal in Computer Virology*, 8(4):141–149, 2012. doi: 10.1007/s11416-012-0165-0. URL http://dx.doi.org/10.1007/s11416-012-0165-0.

[46] Danial Javaheri, Mehdi Hosseinzadeh, and Amir Masoud Rahmani. Detection and elimination of spyware and ransomware by intercepting kernel-level system routines. *IEEE Access*, 6:78321–78332, 2018. doi: 10.1109/access.2018.2884964. URL http://dx.doi.org/10.1109/access.2018.2884964.

[47] Kesav Kancherla, John Donahue, and Srinivas Mukkamala. Packer identification using byte plot and markov plot. *Journal of Computer Virology and Hacking Techniques*, 12(2):101–111, 2016. doi: 10.1007/s11416-015-0249-8. URL http://dx.doi.org/10.1007/s11416-015-0249-8.

[48] Murat Kantarcıoğlu, Bowei Xi, and Chris Clifton. Classifier evaluation and attribute selection against active adversaries. *Data Mining and Knowledge Discovery*, 22(1):291–335, 2011. doi: 10.1007/s10618-010-0197-3. URL http://dx.doi.org/10.1007/s10618-010-0197-3.

[49] Stefan Kiltz, Andreas Lang, and Jana Dittmann. Malware: specialized trojan horse. In *Cyber Warfare and Cyber Terrorism*, pages 154–160. IGI Global, 2007.

[50] Stephanie Kirchgaessner, Paul Lewis, David Pegg, Sam Cutler, Nina Lakhani, and Michael Safi. Revealed: leak uncovers global abuse of cyber-surveillance weapon, 2021. The Guardian. Sun 18 Jul 2021, online.

[51] Peter Knight. Iloveyou: Viruses, paranoia, and the environment of risk. *The Sociological Review*, 48(2_suppl):17–30, 2000. doi: 10.1111/j.1467-954x.2000.tb03518.x. URL `http://dx.doi.org/10.1111/j.1467-954x.2000.tb03518.x`.

[52] J Zico Kolter and Marcus A Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7(12), 2006.

[53] Simon Kramer and Julian C Bradfield. A general definition of malware. *Journal in computer virology*, 6(2):105–114, 2010. doi: 10.1007/s11416-009-0137-1. URL `http://dx.doi.org/10.1007/s11416-009-0137-1`.

[54] Sushil Kumar et al. An emerging threat fileless malware: a survey and research challenges. *Cybersecurity*, 3(1):1–12, 2020. doi: 10.1186/s42400-019-0043-x. URL `http://dx.doi.org/10.1186/s42400-019-0043-x`.

[55] David Kushner. The real story of stuxnet. *ieee Spectrum*, 50(3):48–53, 2013. doi: 10.1109/mspec.2013.6471059. URL `http://dx.doi.org/10.1109/mspec.2013.6471059`.

[56] Daniel T Larose and Chantal D Larose. *Discovering knowledge in data: an introduction to data mining*, volume 4. John Wiley & Sons, 2014.

[57] Pavel Laskov et al. Practical evasion of a learning-based classifier: A case study. In *2014 IEEE symposium on security and privacy*, pages 197–211. IEEE, 2014. doi: 10.1109/sp.2014.20. URL `http://dx.doi.org/10.1109/sp.2014.20`.

[58] Jinghua Li, Xiaoyan Liu, Huixiang Zhang, and Dejun Mu. A scalable cloud-based android app repackaging detection framework. In *Green, Pervasive, and Cloud Computing*, pages 113–125. Springer, 2016. doi: 10.1007/978-3-319-39077-2_8. URL `http://dx.doi.org/10.1007/978-3-319-39077-2_8`.

[59] Wei-Jen Li, Ke Wang, Salvatore J Stolfo, and Benjamin Herzog. Fileprints: Identifying file types by n-gram analysis. In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, pages 64–71. IEEE, 2005. doi: 10.1109/iaw.2005.1495935. URL `http://dx.doi.org/10.1109/iaw.2005.1495935`.

[60] Xufang Li, Peter KK Loh, and Freddy Tan. Mechanisms of polymorphic and metamorphic viruses. In *2011 European intelligence and security informatics conference*, pages 149–154. IEEE, 2011. doi: 10.1109/eisic.2011.77. URL `http://dx.doi.org/10.1109/eisic.2011.77`.

[61] Shuang Liang and Xiaojiang Du. Permission-combination-based scheme for android mobile malware detection. In *2014 IEEE international conference on communications (ICC)*, pages 2301–2306. IEEE, 2014. doi: 10.1109/icc.2014.6883666. URL `http://dx.doi.org/10.1109/icc.2014.6883666`.

[62] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory.* John Wiley & Sons, 2014.

[63] Wei Liu and Sanjay Chawla. Mining adversarial patterns via regularized loss minimization. *Machine learning*, 81(1):69–83, 2010. doi: 10.1007/s10994-010-5199-2. URL `http://dx.doi.org/10.1007/s10994-010-5199-2`.

[64] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007. doi: 10.1109/msp.2007.48. URL `http://dx.doi.org/10.1109/msp.2007.48`.

[65] Brent Maheux. Assessing the intentions and timing of malware. *Technology Innovation Management Review*, 4(11), 2014.

[66] Alejandro Martín, Alejandro Calleja, Héctor D Menéndez, Juan Tapiador, and David Camacho. Adroit: Android malware detection using meta-information. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2016. doi: 10.1109/ssci.2016.7849904. URL `http://dx.doi.org/10.1109/ssci.2016.7849904`.

[67] Alejandro Martín, Héctor D Menéndez, and David Camacho. String-based malware detection for android environments. In *International Symposium on Intelligent and Distributed Computing*, pages 99–108. Springer, 2016. doi: 10.1007/978-3-319-48829-5_10. URL `http://dx.doi.org/10.1007/978-3-319-48829-5_10`.

[68] Alejandro Martín, Héctor D Menéndez, and David Camacho. Mocdroid: multi-objective evolutionary classifier for android malware detection. *Soft Computing*, 21(24):7405–7415, 2017. doi: 10.1007/s00500-016-2283-y. URL `http://dx.doi.org/10.1007/s00500-016-2283-y`.

[69] Alejandro Martín, Raul Lara-Cabrera, and David Camacho. Android malware detection through hybrid features fusion and ensemble classifiers: the andropytool framework and the omnidroid dataset. *Information Fusion*, 52:128–142, 2019. doi: 10.1016/j.inffus.2018.12.006. URL `http://dx.doi.org/10.1016/j.inffus.2018.12.006`.

[70] Héctor D Menéndez and José Luis Llorente. Mimicking anti-viruses with machine learning and entropy profiles. *Entropy*, 21(5):513, 2019. doi: 10.3390/e21050513. URL `http://dx.doi.org/10.3390/e21050513`.

[71] Héctor D Menéndez, Sukriti Bhattacharya, David Clark, and Earl T Barr. The arms race: Adversarial search defeats entropy used to detect malware. *Expert Systems with Applications*, 118:246–260, 2019. doi: 10.1016/j.eswa.2018.10.011. URL `http://dx.doi.org/10.1016/j.eswa.2018.10.011`.

[72] Héctor D Menéndez, David Clark, and Earl T Barr. Getting ahead of the arms race: Hothousing the coevolution of virustotal with a packer. *Entropy*, 23(4):395, 2021. doi: 10.3390/e23040395. URL `http://dx.doi.org/10.3390/e23040395`.

[73] Savita Mohurle and Manisha Patil. A brief study of wannacry threat: Ransomware attack 2017. *International Journal of Advanced Research in Computer Science*, 8(5):1938–1940, 2017.

[74] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007. doi: 10.1109/acsac.2007.21. URL `http://dx.doi.org/10.1109/acsac.2007.21`.

[75] Carey Nachenberg. Computer virus-antivirus coevolution. *Communications of the ACM*, 40(1):46–51, 1997. doi: 10.1145/242857.242869. URL `http://dx.doi.org/10.1145/242857.242869`.

[76] János Neumann, Arthur W Burks, et al. *Theory of self-reproducing automata*, volume 1102024. University of Illinois press Urbana, 1966. doi: 10.1016/0020-0271(69)90026-6. URL `http://dx.doi.org/10.1016/0020-0271(69)90026-6`.

[77] Umara Noor, Zahid Anwar, Tehmina Amjad, and Kim-Kwang Raymond Choo. A machine learning-based fintech cyber threat attribution framework using high-level indicators of compromise. *Future Generation Computer Systems*, 96:227–242, 2019. doi: 10.1016/j.future.2019.02.013. URL `http://dx.doi.org/10.1016/j.future.2019.02.013`.

[78] Digit Oktavianto and Iqbal Muhardianto. *Cuckoo malware analysis*. Packt Publishing Ltd, 2013.

[79] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, volume 41, page 86, 2009.

[80] Sergio Pastrana and Guillermo Suarez-Tangil. A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth. In *Proceedings of the Internet Measurement Conference*, pages 73–86, 2019.

[81] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *NSDI*, volume 10, page 14, 2010.

[82] Sebastian Poeplau and Jan Gassen. A honeypot for arbitrary malware on usb storage devices. In *2012 7th International Conference on Risks and Security of Internet and Systems (CRiSIS)*, pages 1–8. IEEE, 2012. doi: 10.1109/crisis.2012.6378948. URL `http://dx.doi.org/10.1109/crisis.2012.6378948`.

[83] Marco Prandini and Marco Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[84] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. A semantics-based approach to malware detection. *ACM SIGPLAN Notices*, 42(1):377–388, 2007.

[85] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. Camouflage in malware: from encryption to metamorphism. *International Journal of Computer Science and Network Security*, 12(8):74–83, 2012.

[86] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 49–64, 2015.

[87] Chuangang Ren, Yulong Zhang, Hui Xue, Tao Wei, and Peng Liu. Towards discovering and understanding task hijacking in android. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 945–959, 2015.

[88] Igor Santos, Felix Brezo, Javier Nieves, Yoseba K Penya, Borja Sanz, Carlos Laorden, and Pablo G Bringas. Idea: Opcode-sequence-based malware detection. In *International Symposium on Engineering Secure Software and Systems*, pages 35–43. Springer, 2010. doi: 10.1007/978-3-642-11747-3_3. URL http://dx.doi.org/10.1007/978-3-642-11747-3_3.

[89] Igor Santos, Javier Nieves, and Pablo G Bringas. Semi-supervised learning for unknown malware detection. In *International Symposium on Distributed Computing and Artificial Intelligence*, pages 415–422. Springer, 2011. doi: 10.1007/978-3-642-19934-9_53. URL http://dx.doi.org/10.1007/978-3-642-19934-9_53.

[90] A-D Schmidt, Rainer Bye, H-G Schmidt, Jan Clausen, Osman Kiraz, Kamer A Yuksel, Seyit Ahmet Camtepe, and Sahin Albayrak. Static analysis of executables for collaborative malware detection on android. In *2009 IEEE International Conference on Communications*, pages 1–5. IEEE, 2009. doi: 10.1109/icc.2009.5199486. URL http://dx.doi.org/10.1109/icc.2009.5199486.

[91] E Eugene Schultz. Where have the worms and viruses gone?—new trends in malware. *Computer Fraud & Security*, 2006(7):4–8, 2006. doi: 10.1016/s1361-3723(06)70398-0. URL http://dx.doi.org/10.1016/s1361-3723(06)70398-0.

[92] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 38–49. IEEE, 2000. doi: 10.1109/secpri.2001.924286. URL http://dx.doi.org/10.1109/secpri.2001.924286.

[93] James Scott. Signature based malware detection is dead. *Institute for Critical Infrastructure Technology*, 2017.

[94] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International symposium on research in attacks, intrusions, and defenses*, pages 230–253. Springer, 2016. doi: 10.1007/978-3-319-45719-2_11. URL http://dx.doi.org/10.1007/978-3-319-45719-2_11.

[95] Sergiu Sechel. A comparative assessment of obfuscated ransomware detection methods. *Informatica Economica*, 23(2):45–62, 2019. doi:

10.12948/issn14531305/23.2.2019.05. URL `http://dx.doi.org/10.12948/issn14531305/23.2.2019.05`.

[96] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software.* no starch press, 2012. doi: 10.1016/j.cose.2012.05.004. URL `http://dx.doi.org/10.1016/j.cose.2012.05.004`.

[97] Ivan Sorokin. Comparing files using structural entropy. *Journal in computer virology*, 7(4):259–265, 2011. doi: 10.1007/s11416-011-0153-9. URL `http://dx.doi.org/10.1007/s11416-011-0153-9`.

[98] Salvatore J Stolfo, Ke Wang, and Wei-Jen Li. Towards stealthy malware detection. In *Malware Detection*, pages 231–249. Springer, 2007. doi: 10.1007/978-0-387-44599-1_11. URL `http://dx.doi.org/10.1007/978-0-387-44599-1_11`.

[99] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys & Tutorials*, 16(2):961–987, 2013. doi: 10.1109/surv.2013.101613.00077. URL `http://dx.doi.org/10.1109/surv.2013.101613.00077`.

[100] Guillermo Suarez-Tangil, Juan E Tapiador, Flavio Lombardi, and Roberto Di Pietro. Alterdroid: differential fault analysis of obfuscated smartphone malware. *IEEE Transactions on Mobile Computing*, 15(4):789–802, 2015. doi: 10.1109/tmc.2015.2444847. URL `http://dx.doi.org/10.1109/tmc.2015.2444847`.

[101] S Momina Tabish, M Zubair Shafiq, and Muddassar Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, pages 23–31, 2009. doi: 10.1145/1599272.1599278. URL `http://dx.doi.org/10.1145/1599272.1599278`.

[102] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: automatic reconstruction of android malware behaviors. In *Ndss*, 2015.

[103] Acar Tamersoy, Kevin Roundy, and Duen Horng Chau. Guilt by association: large scale malware detection by mining file-relation graphs. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1524–1533, 2014.

[104] F Tchakounté and P Dayang. System calls analysis of malwares on android. *International Journal of Science and Technology*, 2(9):669–674, 2013.

[105] Victor Van Der Veen, Herbert Bos, and Christian Rossow. Dynamic analysis of android malware. *Internet & Web Technology Master thesis, VU University Amsterdam*, 2013.

[106] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007. doi: 10.1109/msp.2007.45. URL `http://dx.doi.org/10.1109/msp.2007.45`.

[107] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[108] Peidai Xie, Xicheng Lu, Yongjun Wang, Jinshu Su, and Meijian Li. An automatic approach to detect anti-debugging in malware analysis. In *International Conference on Trustworthy Computing and Services*, pages 436–442. Springer, 2012. doi: 10.1007/978-3-642-35795-4_55. URL `http://dx.doi.org/10.1007/978-3-642-35795-4_55`.

[109] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers. In *Proceedings of the 2016 network and distributed systems symposium*, volume 10, 2016.

[110] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 732–744, 2015. doi: 10.1145/2810103.2813663. URL `http://dx.doi.org/10.1145/2810103.2813663`.

[111] Yanfang Ye, Tao Li, Qingshan Jiang, and Youyu Wang. Cimds: adapting postprocessing techniques of associative classification for malware detection. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 40(3):298–307, 2010. doi: 10.1109/tsmcc.2009.2037978. URL `http://dx.doi.org/10.1109/tsmcc.2009.2037978`.

[112] Suleiman Y Yerima, Sakir Sezer, Gavin McWilliams, and Igor Muttik. A new android malware detection approach using bayesian classification. In *2013 IEEE 27th international conference on advanced information networking and applications (AINA)*, pages 121–128. IEEE, 2013. doi: 10.1109/aina.2013.88. URL `http://dx.doi.org/10.1109/aina.2013.88`.

[113] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010. doi: 10.1109/bwcca.2010.85. URL `http://dx.doi.org/10.1109/bwcca.2010.85`.

[114] Lenny Zeltser. Reverse engineering malware. *Retrieved June*, 13:2010, 2001.

[115] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE, 2012. doi: 10.1109/sp.2012.16. URL `http://dx.doi.org/10.1109/sp.2012.16`.

[116] Yan Zhou, Murat Kantarcioglu, Bhavani Thuraisingham, and Bowei Xi. Adversarial support vector machine learning. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1059–1067, 2012.

[117] Yan Zhou, Murat Kantarcioglu, and Bowei Xi. A survey of game theoretic approach for adversarial machine learning. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 9(3):e1259, 2019.