POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

# Toward a methodology for malware analysis and characterization for Machine Learning application

**Supervisor**
prof. Antonio Lioy
prof. Andrea Atzeni

**Candidate**
Andrea Sindoni

April 2023

# Summary

In the last decades malware has been one of the major threats for IT systems, targeting both end users and organizations. Year after year malware samples evolve, showing new mechanisms to take advantage of their victims and developing new techniques to avoid detection. The analysis process is a fundamental task needed to perform both identification, i.e. labelling a program as benign or malicious, and family characterization, which means understanding which family a certain sample belongs to. A malware family is a group of samples that share very common characteristics or that have been developed by the same malicious actor. Being able to correctly characterize a sample will provide useful information to victims of malware attacks, since there could be known countermeasures to the actions a certain family performs as well as understanding which is the malicious actor behind the attack. This thesis focuses on the development of an analysis and characterization methodology, trying to leverage on already developed tools that are able to extract representative information, i.e. features, from samples and trying to automate the extraction process as much as possible. Features could later be used to perform characterization by preparing it to become a valid input for a Machine Learning system.

In a first phase I tried to get an overview of the current threat landscape, examining reports of various Antivirus companies, such as Kaspersky, in order to understand which are the most widespread families and what Operating Systems are mostly affected by malware. After having collected this information, I selected a pool of families that target Windows and Android Operating Systems, which are the ones mostly used on desktop and mobile devices.

Then, I performed a study on the state-of-the-art techniques and tools that nowadays are used to perform malware analysis, both to become able to build a safe analysis environment and to perform a selection of tools that can be used to carry the analysis process. I focused both on static and dynamic techniques trying different tools, both manual ones and fully automated, such as automatic online sandboxes, selecting those that were capable of extracting relevant information as well as being suitable for automation.

Once this selection was made, I started collecting samples belonging to the selected families, looking for available malware datasets and repositories, like VirusShare and MalwareBazaar, and combining them together to create my own collection. Two different collections were built: a Windows one, consisting of ten thousand executables from eight families, and an Android one, consisting of six hundred applications from six different families.

Later, I focused on the development of an analysis methodology, building a system that is capable of automatically extract features from samples, applying both static and dynamic analysis techniques studied in the previous steps. Despite there are many frameworks that already provide a safe environment, fully equipped with tools to perform analysis, they are often intended to be used manually, studying each sample one by one, meanwhile I wanted automate their usage as much as possible since they had to be applied on a large number of samples. To this end, I decided to build my analysis system with a static and a dynamic analyzer. The former has been developed using two Python libraries: *Pefile* and *Droidlysis*, which are employed to extract static features. The latter has been built using the Cuckoo sandbox and automating its functionalities. I was able

to extract many kinds of information contained into my samples, but I had to discard some types of features, such as strings and IP addresses, since they were not perfectly suitable for a Machine Learning system. In the end, I selected four kind of features for the Windows environment: *sections' bytes, imported APIs, sections' entropy* and *API calls count.* Three features were selected for Android: *permissions, code properties, API calls count.*

Each sample has been processed in order to extract the selected features and then prepare them in a series of datasets that could be later used as input for training a ML system. The extracted information has also been reviewed to find any similarities and differences between the features extracted from samples. Overall, every feature from the ones we selected showed significant differences between samples belonging to different families. Looking at the Windows environment, sections' entropy and the number of API calls seem to be the two features that mostly characterize each family. The latter in particular, showed how some families heavily stress the OS, making a huge number of API calls (i.e. Virlock family averages four thousands calls per minute on some APIs). On the other side, the features extracted from Android samples showed interesting differences on the kind of actions that each family tries to perform. The selected features give a general description of what are the potential actions performed by samples. I focused on harmful actions, and the results showed how some of them are shared among every family (i.e. almost 65% of samples of each family collects device's information), while others are more specific to some particular family (i.e. 85% of Svpeng samples writes system settings and disables the lock screen mechanism).

For what concerns future developments, this system could be extended, adding more complex types of features that could require a proper encoding in order to be handled, or by extending the file formats supported, since supporting only Android Packages (APKs) and Portable Executables (PEs) is a strong limitation, considering the many possible flavours in which malware usually appears.

# Contents

# Chapter 1

# Introduction

Year after year IT technologies keep evolving, becoming more powerful and offering more features to their users, infusing into our lives and becoming a crucial part of any kind of environment, from complex systems to simple things that surround us every day. As the technology improves, its threats evolve as well, stressing every possible system and trying to find the most subtle vulnerability. The term *malware*, a short for malicious software, refers to any kind of software that tries to perform malicious activities once running on a system. Possible examples of such activities could be: spying, take control of a system or stealing credentials. Malware can cause huge damages, having a significant impact not only on individuals but also on organizations. In the last year, more than 5.5 billion malware attacks have been detected [1]. Fig. 1.1 reports the global volume of malware samples for each month of the years 2021-2022.

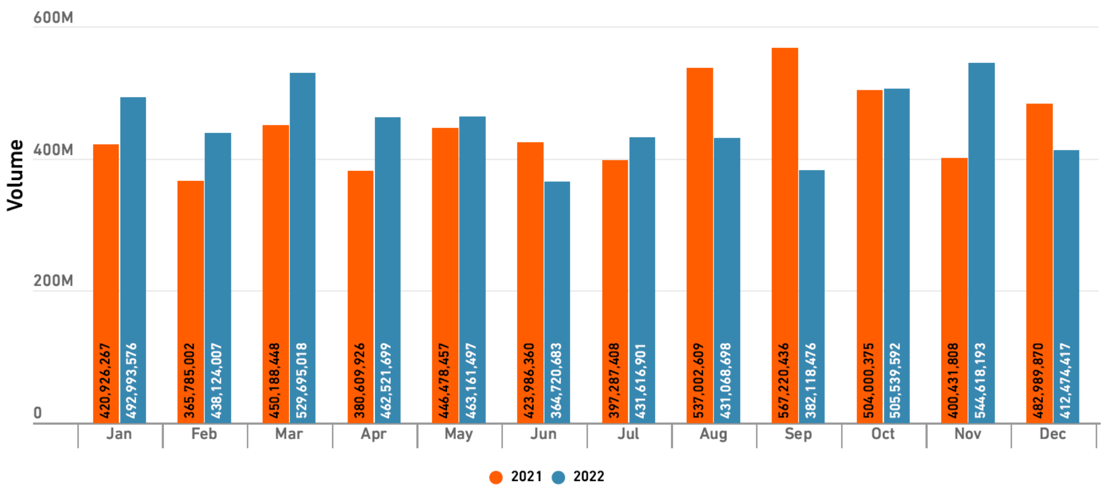


Figure 1.1. Global malware volume comparison between 2021 and 2022. Image taken from SonicWall report [1].

As time passes, malware samples continue to evolve, constantly showing new mechanisms to exploit vulnerable systems and to prevent detection. This means that defense systems that are in charge of preventing malware infections need to be up to date in order to keep the pace of such a menace. To this end, the ability to perform identification and characterization assumes a crucial role. Identification means being able to assert whether a sample is malicious or benign. For example, identification is always performed by the Google Play Store when a new application is published, since that application can be downloaded by billion of devices therefore it must be checked before it becomes available to anyone. Characterization refers to the ability of distinguish between samples belonging to different families. A malware family is defined as a group of samples which shares very common characteristics, being probably developed and spread by

the same threat actor. Characterization can provide useful information, revealing who is the actor behind an attack and maybe, in some particular cases, it gives the chance to take proper countermeasures against the actions that a malware has performed. In order to become able to identify characterize, analysis must be performed, since it is needed to investigate on malicious behaviors and understand how to distinguish between benign and malicious actions, as well as to find peculiar characteristics that may be common for a certain families. Analysis can be carried out through different techniques, depending on its purpose and the way it is performed, hence what the analyst is looking for and how it achieves it. Generally speaking, there are two main approaches: *static analysis* and *dynamic analysis*. Static analysis techniques are usually faster and less expensive, both in the time and the amount of resources required, but they are easily counterable and malware may hide some aspects that will be revealed only upon execution. On the contrary, dynamic techniques require a substantial amount of resources, both computational and in time, but any static countermeasure will be bypassed since the sample is actually executed and monitored. Of course, dynamic analysis can be countered as well, since the analysis environment can be detected and the sample will suspend any suspicious activity.
As of the day of this writing, various tools and techniques have been already developed to study malware samples, from simple manual tools such as Ghidra [2], which is a disassembler/decompiler that allows to reverse engineering executables, to more sophisticated instruments such as Remnux [3], which is a Linux distribution fully equipped with many kinds of analysis tools. These instruments allows to investigate on many malware-related aspects. Some of them are just property extractors (i.e. they parse the provided file to extract static information), meanwhile some others can trace actions on the system, like monitoring opened files, function calls and so on. For example Autoruns [4] permits to check for programs that runs at boot, revealing mechanisms that malware may use to obtain persistence. Such tools are suitable for the contest of manual analysis, that can be particularly useful when studying single samples which may belong to new families or just be a new variant of an already known family. Manual analysis requires the analyst to be highly skilled, since it has to deal with many tools such as decompilers and debuggers to study any possible behavior of the sample under analysis, as well as requiring deep knowledge of the Operating System for which the sample has been created. Manual analysis takes definitely a considerable amount of time, in some cases reversing a single sample could require even a week of work. This means that if the analysis has to be applied to a large scale of samples, for example if we want to provide training data to a Machine Learning (ML) system, which has been built to fulfill the task of characterization/identification, a manual approach becomes unfeasible and it is necessary to automate as much as possible the analysis process. Machine Learning allows to perform classification on a large number of samples in a very short time, but in order to build a valid classifier, this has to be trained using a proper dataset of samples. Delivering a dataset of actual executables is quite risky though, since if samples are mistakenly executed the whole environment will be compromised. Other than that, providing only samples just as they are, leaves the duty of feature extraction (i.e. information that can be used to perform classification) to the ML system which shall be equipped with a property extractor. Fig. 1.2 reports an high-level view of the structure of a classification system.

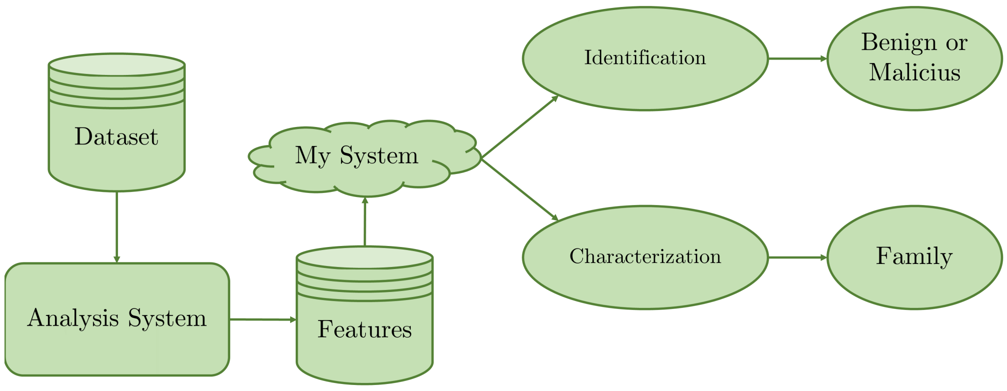


Figure 1.2. General structure of a classification system.

Previous researches have developed ML systems in charge of performing identification through malware features [5, 6], or characterization using one single feature such as system calls [7]. These systems use as sources, pre-built datasets of features, or they provide a complete system composed of analyzer and classifier. Our work wants to focus on the first part (i.e. the analyzer) in order to develop an analysis methodology that aims to extract different types of features, which can be used one by one or combined, providing more flexibility compared to those systems developed in previous researches. In addition to that, most of these systems points on identification rather than characterization, which still is an open problem.
This research focuses on the design of an analysis methodology, developing an automatic analysis system which, leveraging on well-known malware analysis tools, is capable of extracting representative information from malware samples, giving the possibility to customize the kind of features that will be extracted and also to apply analysis on large datasets. Once these features are extracted, the systems prepares them into packages that can be used as input for a ML system.
Chapter two will introduce some background concepts, which are needed to understand the topics covered in this thesis. It gives some definitions of what is malware and what are common malware types as well as some general behavioral characteristics, such as how malware obtains persistence and some examples of anti-analysis techniques. Last sections of this chapter will also introduce some Operating Systems elements that are commonly involved when studying malware's actions.
The third chapter illustrates the state-of-the-art of malware analysis, presenting various tools and analysis techniques that are employed to carry the analysis process, covering both manual and automatic tools. Chapter four will present our two malware collections, containing Windows and Android samples respectively, explaining how these have been built, from the selection of families contained, to the sources that have been used to collect desired samples. Chapter five, which can be considered the core of this research, will introduce the analysis methodology design, presenting our own analysis system, which is depicted in Fig. 1.3, explaining how each of its components works as well as reporting the information that the systems is able to extract.

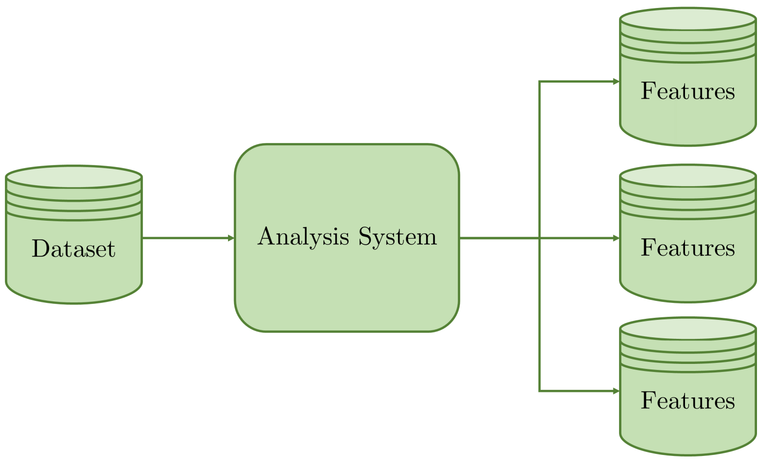


Figure 1.3. High-level view of the analysis system.

Chapter six will present the obtained results, providing insights on the datasets of features that have been created by submitting our collections to the analysis system. Other than that, this chapter contains a series of statistics regarding these features, outlining similarities and differences between families.

# Chapter 2

# Background

## 2.1 What is malware?

Malware stands for "malicious software", referring to any kind of software that aims to perform malicious activity such as spying, steal information or take control of a system. Various companies and organizations have given more specific definitions:

*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.* - **NIST** [8]

*A type of computer program designed to infect a legitimate user's computer and inflict harm on it in multiple ways.* - **Kaspersky** [9]

*Any intrusive software developed by cybercriminals to steal data and damage or destroy computers and computer systems.* - **CISCO** [10]

### 2.1.1 Why is malware used?

Malicious actors use malware for different purposes. Usually the most attractive targets are large companies, but often single users are attacked as well, especially with the spread of mobile malware. The main reasons that lead to the use of malware are [11]:

- **Mining cryptocurrencies**: victim's computer resources are abused to generate or mine cryptocurrencies.
- **Data theft**: data is stolen to sell it to other criminals or to commit identity theft.
- **Asking for ransom**: access to victim's data is denied and a payment is demanded to have it back.
- **Espionage**: this can be done even by companies to collect secrets from their competitors.
- **Law enforcement**: government authorities may use malware to investigate and collect data on suspects and use it later for their investigation.
- **DDOS attack**: malware can be deployed to create a botnet and coordinate it to perform a DDOS attack.

### 2.1.2 Malware types

Malware can be classified in different typologies. This distinction is mainly based on the malware behavior, hence what it tries to achieve once it is running on the infected system.
The main known types are [12]:

- **Virus**: it injects its code into other files and tries to spread to other hosts. Viruses may even modify the copies that will be attached to other files or hosts. There are different spreading vectors such as email attachments, network files, etc.
- **Trojan**: generally it looks like a benign application like a game, while performing malicious tasks in the background. Trojans are mainly employed to steal victim's sensitive data (i.e. banking credentials). Another variant is the RAT (Remote Access Trojan) which enables a backdoor to provide remote access on the compromised system.
- **Ransomware**: denies access to user's files, demanding payment to have them back. The denial can be done through file encryption (Crypto ransomware) or blocking system functionalities (Locker ransomware) [13].
- **Worm**: it replicates itself, trying to saturate victim's resources (i.e. bandwidth, disk space) and then spreading to other systems over the network.
- **Rootkit**: provides a set of tools that can be used to harm the system. It is usually used by other malwares. Rootkits employ several techniques in order to conceal their existence to the system (DLL injection, system table hooking etc.) and there are different typologies, depending on which privilege level they operate [14].
- **Spyware**: it spies the user, keeping track of whatever she does and sends reports to the attacker. The information that a spyware collects is very various: password, browser activity, credit card details etc.
- **Scareware**: generally it does not harm the system but just informs the user that is device has been infected, then it tries to sell an antivirus to remove itself.
- **Cryptominers**: they use computer resources to mine cryptocurrencies for the attacker.
- **Adware**: shows advertisements to the user, usually it is not harmful for the system.

### 2.1.3 Malware families

Another important distinction that has been made is the one between families. A malware family is a group of malware samples which shares a large portion of code, showing a very similar behavior, and that have been probably developed by the same author. Samples belonging to the same family may have some differences, thus leading to a mismatch when comparing file hashes, but the overall behavior will be the same. Family classification is not an easy job, since it requires to understand what are the important features that remains similar across multiple samples. There are many famous families that have been in the wild for quite a long time, such as Zeus (2007), Emotet (2014), WannaCry (2017) and that can still be found today, as well as new families that have made their name in the recent years, for example the Lockbit family, for which a new version was deployed in June 2022 [15].

## 2.2 Malware behaviors

Generally speaking, malware employs several techniques in order to achieve its purpose. There are some aspects that are common even to different typologies and some that are more specific and tend to be exclusive to some types. Some of these are shown in Table 2.1.

| **Common actions** | **Specific actions** |
|---|---|
| Gaining persistence | Stealing information (Trojan, Ransomware) |
| Privilege escalation | Denying service (Worm, Ransomware) |
| Communication with a C&C server | Stealing resources (Miners) |
| Anti-analysis techniques | Spreading (Virus, worm) |
| | Deceiving the user (Trojan, Scareware) |
| | Annoying the user (Adware) |

Table 2.1. Common and specific malware actions.

### 2.2.1 Gaining persistence

Obtaining persistence on the infected system is an important step for the malware, probably the most important for some typologies. It means obtaining access to a system across restarts or other kind of interruptions. Often it requires high privileges since the malware has to change configuration settings. It is usually achieved using OS's internal structures like the registry on Windows or, in the case of a ransomware, by deleting backups, preventing the user from restoring the system to a point where the malware wasn't there.

### 2.2.2 Privilege escalation

Obtaining the highest privilege on the system is a crucial task for malware since it allows to perform any kind of operation without asking its victim. It is usually done by using software or system vulnerabilities or by annoying the user until she grants it. Many malware families have been known for using famous expoloits such as EternalBlue[16], used by WannaCry and Trickbot families, or the SpoolFool[17] used by the Lockbit family [18]. As previously mentioned, sometimes the malware uses an external C&C server to obtain these exploits, since it could be running on an outdated versions of an OS that contains known vulnerabilities and for which there are exploits available, so it do not even require much work in those cases, malware authors just grabs what is available online. Obtaining high priviliges can be very trivial sometimes [19].

### 2.2.3 Communication with a C&C server

Communication with a C&C server serves for multiple purpose. The client (the malware) generally sends reports containing useful information about the system (location, OS version, installed programs etc.). The server may use this information to send specific payloads to the malware. For example, if the malware is running on a OS version that contains a known vulnerability, the server might sends the payload to exploit it. The server can also send whole modules or components that the malware needs to run. In some cases (i.e. WannaCry), the server may also act as a kill switch sending a command to the malware that will stop its activities. Communication is usually encrypted and to avoid detection some families use a DGA (Domain Generation Algorithm). This is used to generate a large number of domains that will be contacted until one of them actually responds. The domain generation is based on three elements: a seed, an element that changes over time and some top level domains [20]. The domains must be predictable on both client and server side.

### 2.2.4 Anti-analysis techniques

Malware authors employs different countermeasure to protect their samples. These countermeasures aim at complicating the analysis process, hence making the detection of such samples more difficult. These techniques will be discussed deeply in Sec. 2.4

## 2.3 Malware analysis

The process of malware analysis is the main focus of this thesis. It is a fundamental task that aims to understand what are the peculiarities of the sample that is being analyzed. It is a time consuming activities, since malware authors don't want their samples to be easily analyzable, employing many countermeasures to make this process as complex as possible. Of course this task cannot always be performed manually, considering that in some cases analyzing even a singe sample could require days. To help with this there are many tools that come to hand, which are able to automatically extract useful information, saving a considerable amount of time.
When we want to perform software analysis there are three strategies: *white-box, black-box* and *grey-box*. *White-box* means that we would have full knowledge of the software under analysis, having full access to any resource. *Black-box* is the complete opposite, we are examining a program without knowing anything about it. *Grey-box* is a combination of the previous two, we have a partial knowledge. When speaking about malware analysis, the only approach available is of course black box.
Generally malware analysis, or maybe it would be better to say software analysis, can be carried on using two approaches: *static* and *dynamic* [21, 22]. This section will briefly describe what these methods actually mean and how they are usually performed, leaving a deeper insight on what is used for malware analysis for the next chapter.

### 2.3.1 Static analysis

Static analysis consists in analyzing the software without running it, trying to extract meaningful information that could give hints on what the program does. Usually it is carried on by using automatic tools such as disassemblers, that extract the assembly code from the binary, or decompilers, which try to reconstruct the original source code. These tools require lots of manual work, since the software analyst has to understand what the decompiled (or disassembled) code means. Since this task can become a cumbersome in some situations, there are static analysis tools such as Ghidra or IDA that provides also a CFG (Control Flow Graph) extractor, String extractor and other useful instruments.

### 2.3.2 Dynamic analysis

Dynamic analysis, also known as behavioral analysis, is based upon examining the software while it is running, monitoring and observing its behavior. When the software under analysis is a malware it is also suggested to perform the analysis in an isolated environment to avoid any damage. Details on how to properly isolate the malware will be discussed later. There are different approaches that are commonly used such as function calls monitoring, taint analysis etc.

### 2.3.3 Static and dynamic analysis tools

Performing software analysis could require a considerable amount of time, especially when the software under analysis is a malware, whose authors do not want to be easily analyzable, taking any kind of countermeasure to complicate this task. Luckily, over time a wide range of tools have been developed. Most of these tools provides nice features that are able to extract useful information from the software, speeding up the whole analysis process.
Some of the most famous tools are listed in Tab. 2.2, they are not especially meant for malware analysis but for more generic black-box software analysis. The tools that have been explicitely designed for the analysis of malware will be covered in the next chapter.

| Tool | Features | Platform |
|---|---|---|
| Ghidra | Decompiler, Disassembler, CFG, etc. | Any |
| IDA Pro | Decompiler, Disassembler, CFG, etc. | Any |
| Binary ninja | Decompiler, Disassembler etc. | Any |
| Apktool | Decompiler, Debugger, Resource extractor | Android |
| OllyDBG | Debugger | Windows |
| GDB | Debugger | Linux |
| Radare2 | Disassembler, Debugger, Hexeditor | Linux |

Table 2.2. Software analysis tools.

### 2.3.4 Static vs Dynamic

When performing analysis both approaches are needed since each of them covers aspects that are fundamental to understand what a malware does. For example a malware may contain some hidden code or module that will not be detected when using only static analysis. Generally speaking, static analysis is a much faster approach that is able, by the means of some tools, to extract useful information in a very short time, giving a first insight on what the malware does, so it is usually performed as first step. On the other side, dynamic analysis takes a considerable amount of time to be performed and also require to setup a strong isolated environment to avoid any damage to the analysis environment, but it grants the possibility of really understanding what is being executed and what the malware, or more generically the software, is doing. Each of these two techniques has its own advantages and disadvantages, the main ones are reported in Table 2.3.

| Static | Dynamic |
|---|---|
| Requires less resources | CPU expensive |
| Fast | Slow |
| Examines malware without executing it | Malware gets executed |
| All the static information can be extracted and analyzed | Only what is executed is analyzed |
| Ineffective against packed malware | Once ran the packed code is extracted and it can be analyzed |

Table 2.3. Differences between static and dynamic analysis.

## 2.4 Countering Analysis

Malware uses various techniques to make the analysis process as complex as possible. These techniques are used not only by malicious actors, but also by legitimate ones with the purpose of protecting intellectual property. The process of countering analysis is very common in malware. Older studies [23] already showed how, even back then, almost 90% of malware uses at least one anti-analysis technique.
There are several methods that are commonly used and that have already been widely described [21, 24, 22, 25], these are presented in the following sections.

### 2.4.1 Countering static analysis

**1. Dead code insertion**
It consists in inserting instructions that won't be actually executed, so this are ineffective on its behavior but they will change its appearance, making the control flow more complex, thus leading to a waste of time when the malware will be analyzed. In practical terms, this is done by inserting

extra *push* operations followed by a *pop* or by adding the so-called *opaque predicates* which are *if* statements with a complicate condition that will always be either true or false.

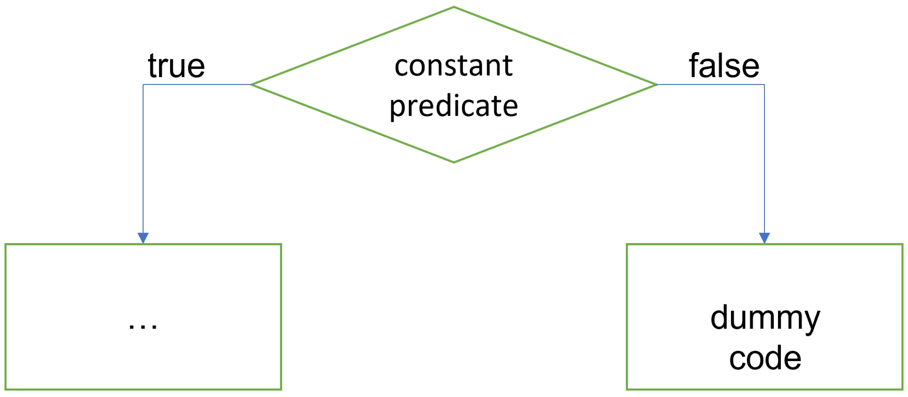


Figure 2.1. Opaque predicate example.

**2. Equivalent Code Replacement**
This techniques substitutes the original instructions with semantically equivalent ones. For example a *mov* can be replaced with a *push/pop* operation or by xoring the value with itself and then perform an *add*. This brings two advantages: a simple instruction may be changed in multiple ones and also it becomes easy to generate several variants of the same malware.
**3. Code transposition**
Instruction sequences are reordered without having any impact on the malware behavior, this can be achieved in two ways:
*1.* Using unconditional branches to jump across the shuffled sequences.
*2.* Only independent instructions are reordered. These are actually quite difficult to find but is the hardest to detect.
**4. Code integration**
The malware integrates itself into another program, this technique is also called *process injection* the methods on how to achieve it are gonna be explained better later in Sec. 2.5.1. Basically it is not longer the main executable that runs the malicious code, but this is attached to benign processes (i.e. by means of a library) that will execute it.
**5. Packing**
This is probably the most advanced and sophisticated obfuscation technique. A packed malware is compressed or encrypted, hence hiding its functionalities. Once executed it unpacks itself and executes the unpacked code. The process of packing can be done or by a simple encryption of the payload or by using some packer such as UPX, UPAC etc.. Other than being a counter analysis technique, packing is mainly used to avoid detection by automatic detector (i.e. AVs).

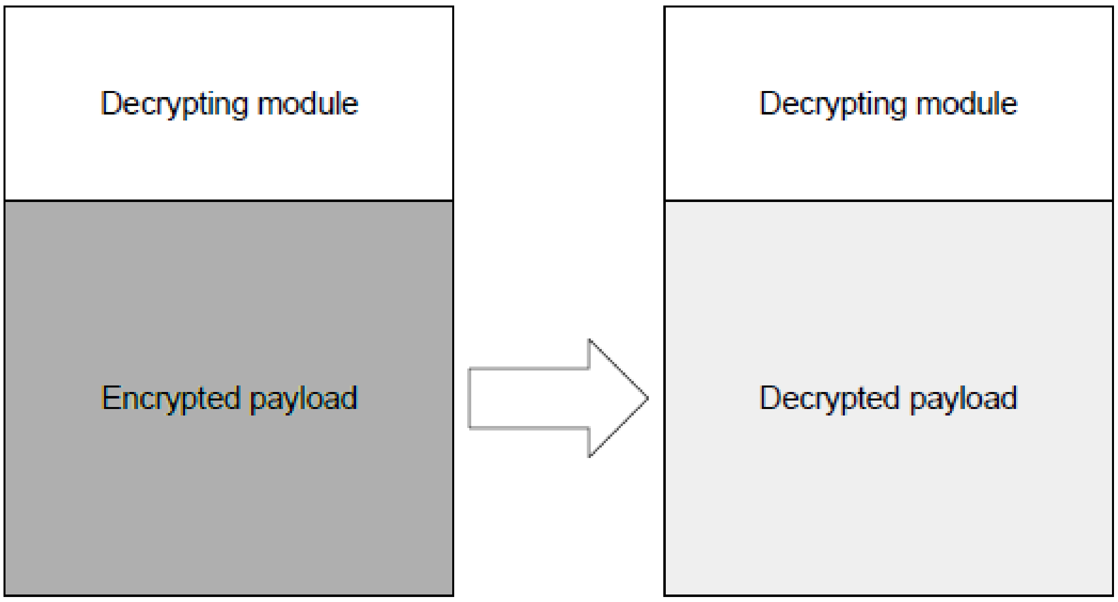


Figure 2.2. Packing example

The main packing techniques are:

- **Encryption**: the malware comes with encryption/decryption module. Usually encryption is performed each time with a different key, but the decryptor remains the same.
- **Oligomorphism**: it overcomes the limit of a single decryptor, gaining the ability to produce different variants which are all semantically equivalent. The main drawback is that only a limited number of decryptors can be made so a detection system can be instructed to detect every possible variant.
- **Polymorphism**: it is an advanced version of oligomorphism. This time unlimited decryptors can be created, hence producing a an unlimited number of variants. In some cases multiple layers of decryption are used. This can still be detecting applying a signature matching at runtime.
- **Metamorphism**: there is no more a constant body or decryptor since there is no more packing and encryption but the code is transformed dynamically while the malware is running.

### 2.4.2 Countering dynamic analysis

Most of the malwares are capable of detecting if they are running in a virtualized environment, which will probably mean that they are being analyzed. If the analysis environment is detected, malware will try to hide itself, by avoiding any suspicious activity or by not running at all. Some example of how the environment is detected are [26, 21]:

**1. Detecting analysis tool's traces**
Some analysis tools leave fingerprints on the system they are being run. This are searched by malware. [27] reports a code snippet from a malware sample that checks if it is running in VMware:

```
MOV EAX, 564D5868 #VMXh
MOV EBX, 0
MOV ECX, 0A
MOV EDX, 5658 #VX
IN EAX, DX # Check for VMWare
CMP EBX, 564D5868
```

The value *0x0A* tells VMware that a version check wants to be performed. The *"VX"* value refers to a VMware port and the IN instruction is used to read data from a specific hardware device (VMware has extended its usage to implement communication channels). If running in the virtual machine, this instruction will return the value *"VMXh"*.

**2. Detecting debuggers**

The presence of a debugger is always a sign that the sample is being analyzed. For example, one common way to perform this check is to invoke the *ptrace* function that won't work if a debugger is already attached. Analogously, on Windows, there are different API calls that can be used to detect the presence of a debugger such as *CheckRemoteDebuggeRpresent* or by checking the *BeingDebugged* flag.

**3. Time-based detection**

Dynamic analysis brings an overhead, malware can measure how it takes to execute some portions of code.

**4. Detect traces of user activities**

Traces of user's activities: analysis environment are often wiped before testing and studying a sample. Thus the malware looks for data that is usually generated when a real user is using the system such as browser history, recent documents, mouse movement etc.

## 2.5 OS features

All the well known operating systems are targeted by malware. Looking at the desktop environment, Windows results to be both the most common OS worldwide [28] and the preferred malware target [29]. Taking a look at the mobile environment, Android takes the lead as most popular mobile OS (over 70% of the global mobile market) [30], and as reported by Kaspersky: *"Over 98% of mobile banking attacks target Android devices"* [31].

### 2.5.1 Windows

Windows is a complex desktop OS. Malware targets many of its internal structures, trying to achieve persistence or to exploit known vulnerabilities and escalate.
Malwares may come on the system in different formats: PE (Portable Executable), office documents, pdf etc., usually delivered through spam emails or dropped by another malware. They use intensively some of the Windows functionalities, these are briefly explained in the following paragraphs.

#### Windows API

The Windows API is an extensive set of functionalities that are provided to interact with the Windows OS. This can be directly used when writing C or C++ programs. The set of functionalities offered spans through different categories, from base services such as file handling to security, networking, graphics and so on. The amount of functionalities is so high that programmers rarely need to use third party libraries. For what concerns types and notation, Windows program usually differs from standard C, C++ programs. C standard types such as *int, double, long* etc. are not very common. They are replaced by DWORD and WORD which represent respectively 32 and 16 bit integers. The main Windows types are listed in Tab 2.4 [32].

| Type | Description |
|---|---|
| WORD (w) | 16 bit unsigned value |
| DWORD (dw) | 32 bit unsigned value |
| Handles (H) | Reference to an object.<br>This should be manipulated only by Windows API |
| Long Pointer (LP) | Pointer to another type.<br>(i.e. LPSTRING, LPBYTE etc.) |
| Callback | A function that will be called by the Windows API |

Table 2.4. Common Windows types.

### Registry

It is a system hierarchical database used to store configuration information (settings, options etc.). In old Windows versions these information was stored in *.ini* files, the registry was created to provide better performances [32]. Malware mainly uses it to achieve persistence, adding autorun programs (the malware itself). The information stored in the registry is organized in keys, which can be intended as folders. There are five root keys, which contain subkeys, containing values entry (a pair name-value). The two most commonly used root keys are:

- HKEY_LOCAL_MACHINE (HKLM): global machine settings.
- HKEY_LOCAL_CURRENT_USER (HKCU): setting specific to the current logged on user.

The registry can be modified through the Windows API, using the Registry Editor or by *.reg* files.

Listing 2.1. *.reg* file example.

```
[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
"MaliciousValue"="C:\Windows\malicious.exe"
```

### DLLs

Dynamic link libraries are used to share code among multiple applications. A DLL is basically an executable that exports functions. This brings an improvement in terms of memory usage since static libraries would require to be allocated by each process that uses them. Another advantage is that when writing a Windows application (both a benign or a malicious one) we can leverage on DLLs that are by default on a Windows system and avoid to provide them. Generally malware authors use DLLs for three purposes [32]:

1. **Store malicious code**: the malware gets on the system as a DLL, trying to load itself into other processes.
2. **Using Windows DLLs**: they contain functionalities needed to interact with the system and malwares will abuse those functionalities. Looking at the imports could give a good insight on what the malware intends to do.
3. **Using third party DLLs**: they are used to avoid the Windows API. For example a malware can use the Mozilla DLL to perform network operations. A malware can also drop custom DLLs that contain extra functionalities such as encryption.

### System Tables: SSDT, IDT and IAT

System tables are heavily used by rootkits to conceal their existence. The purpose of this table is explained in the Table 2.5:

| Table | Purpose |
|---|---|
| SSDT | Maps syscalls to kernel function addresses |
| IAT | Contains address of functions belonging to a DLL |
| IDT | Contains address of interrupt routines |

Table 2.5. Common and specific malware actions.

The main way to exploit the functionalities offered by these three tables is by using hooking techniques. Hooking is a process of altering system (or API) calls, redirecting them to malicious code. This can be done in both user space (IAT, IDT) and kernel space (SSDT). Hooking techniques are widely used by rootkits.
Hooks on the IDT are the simplest to detect since all the interrupt routines must point to a precise memory location (the one allocated to ntoskrnl.exe DLL). The main techniques [33] can be summarized as follows:

**IAT**.
Each process as its own IAT, hence an hook on this table targets a single process. Hooking the IAT means that when the process calls an API the malicious code will be called instead, this can be done by replacing the address of the legitimate function with the malicious one or by using *inline hooking*. The latter option consists in modifying the actual function by creating a jump to the malicious code, leaving the table unmodified.

**SSDT**.
It holds kernel function addresses and, being in the kernel, a modification of this table will affect every process. In modern versions of Windows there are two tables: one for generic routines and one for graphical ones. Hooking this table is more tricky compared to the IAT since it requires the modification of the Registry and other structures. *Inline hooking* can be used in this context as well, but it is important to execute also the original syscall to preserve kernel integrity.

### Stealth Mechanisms

Malware authors always try to find new tactics and techniques to bypass the checks that are performed both from malware analysis and from automatic detectors (i.e. AVs). A malware needs to perform its operation in the most stealthy way in order to avoid detection. This is usually done by injecting the malware into other legitimate processes. There are various techniques that are commonly employed [24]. We can summarize the stealth techniques in three main categories:

**Process Injection**. Malware hides its code into the memory space of benign processes. This can be performed in various ways such as:

- Classical DLL injection: the malicious DLL path is written in the virtual address space of the target process and another thread is created to ensure its execution.
- Reflective DLL injection: the malicious DLL is loaded from memory, not from the disk
- Process hollowing: instead of loading malicious code, the legitimate one is unmapped from memory and filled by the malicious executable.
- Portable Execute injection: CreateRemoteThread API is used to to copy some shellocode (or a whole PE) into an open process. The malware has to find the relocation table of the target first.
- APC injection: Asynchronous Procedure Calls are used to execute malicious code by attaching it to the APC queue of the target thread.
- Thread execution hijacking: thread execution is hijacked by modifying the EIP register of the target to execute malicious shellcode.

**DLL hijacking**. This mechanism is used to get a target process execute a malicious DLL leveraging the DLL search order.

- DLL search order hijacking: Windows searches for DLL in specific paths. Malware tries to place its DLL, with the same name of a legitimate one, in a place that will be examined before its real location.
- Adaptive DLL hijacking: it is used to execute an external library or PE that is not intended to be ran. The module loader provides the DLL's origin point. This is hijacked to execute the malicious DLL and is achieved in four steps as shown in the image below.
  Insert image here

**File-less**. Sometimes malwares are capable of executing in memory without leaving any disk traces. A file-less malware has to drop some files at first but then it may use some forensic tool (e.g. SDelete) to wipe them out. This is done to minimize the amount of artifacts on the system, making analysis way more complex. It is usually done by malwares that do not care about persistence.

### 2.5.2 Android

Being a mobile Operating system, Android presents a different architecture, providing several functionalities to make applications work. Hence also the attacks that malware authors implement are different. Applications are sandboxed, they have their own user ID and run in separate processes. Each time an application is installed a new Linux user and a private directory are created. Application are usually distributed via the app stores; the main one is the Google Play store, but there are also third party markets where apps can be found.

#### Permissions

An interesting part when looking at an Android application, that could give hints on what the app wants to do once installed and running are the permissions it requires. For example, if an app wants to send SMSs it has to declare the **SEND_SMS** permission. Permission are organized in groups (i.e. the SMS permission groups). A complete list can be found on (insert link here). There are five types of permissions, which are [34]:

- install-time: automatically granted when installed, usually they are prompted to the user before installing.
- normal: allow access to data and actions that present very little risk to user's privacy or other apps.
- signature: an application may define a permission and that can be used by another app, but they must be signed by the same certificate.
- runtime: also known as dangerous permissions, the request is prompted on the screen.
- special: defined by the platform or the OEMs.

In older versions of Android (up to version 6) runtime permissions where asked at install time as well and they cannot be disabled. There are two important permissions that are worth of mention, being intensively used by malwares:

- **RECEIVE_BOOT_COMPLETED**: starts an application at each reboot (after it is manually ran a first time)
- **SYSTEM_ALERT_WINDOW**: it allows to draw arbitrary windows on top of other apps. This is not used only by malicious apps. Usually it is used to create floating widgets (i.e. Facebook Messenger, music players etc.).

- **BIND_ACCESSIBILITY_SERVICE**: this enables Accessibility Service, which is a mechanism created to assist users with disabilities. It enables the capability of performing powerful actions on the device such as clicks, scrolls, lock/unlock the phone and so on. It must be explicitly enabled by the user. It can also perform operation while the screen is off as discovered by [35].

Having the last two permissions, an application would be able to perform any kind of activity on the device such as install another application and grant this one admin privileges.

### Attacks

There are various ways in which malware authors exploit Android functionalities. Most of the attacks aim to deceive and mislead the user, trying to steal its sensitive data (i.e. passwords) without her noticing it. Some of these attacks have been explained, and actually improved, by [35], and can be summarized in:

*A. Clickjacking*
The idea is to "steal" clicks. Basically another page is prompted on the screen but when the user clicks on it the click goes below that page. This is partially countered by some OS security mechanisms that prevents the attacker from knowing when and where the user clicks. To overcome this obstacle this attack was improved in the **Context Aware Clickjacking** where the overlay page is covering everything but the button that the attacker wants to be pushed.

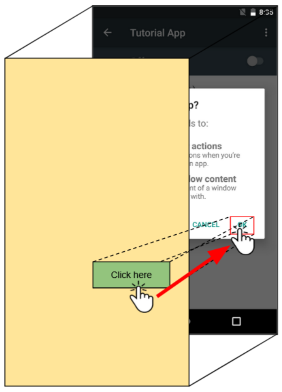


Figure 2.3. Simple clickjacking example, image taken from [36].

*A. Invisible grid*
This attack aims to intercept keystrokes. Several small overlays are created (one for each key on the keyboard). It uses obscured flag FLAG_WATCH _OUTSIDE_TOUCH as a side channel to understand which key is being pressed. This is done by placing overlays one on top of the others, so when the i-th key is pressed the (i-1) previous keys will have this flag set to 1.

# Chapter 3

# State of the art

The analysis process is a fundamental task for the characterization and identification of malware. Despite nowadays most approaches go towards the direction of a Machine Learning (ML) driven method, the manual work is still necessary either to understand what features are important to a ML system or to perform incident response. As previously said, the analysis process can be performed in two ways: *static* and *dynamic*. Both approaches have their pros and cons but they are necessary to have a complete knowledge of what is being analyzed. It is important to note that malware authors employ various techniques to counter both methods and these can make the analysis completely useless if not bypassed in some way.
This chapter will report a series of techniques and tools that are commonly used nowadays to analyse malicious software. These will not be divided into static and dynamic since most tools combine both approaches.

## 3.1 Malware repositories

First of all, before talking about what is actually used to properly analyze malware, we can ask ourselves a simple question: where do we get malware samples?
There are many online resources that provide access to malware samples. For a ML approach, where a huge amount of sample is often required, the best option is to look for an already made dataset. Two example of famous datasets are SOREL [37] and MICROSOFT[38].
If the main purpose is to manually analyze a sample, having a whole dataset could be excessive; in this case a more suitable choice is to download single samples. There are many sites and platforms which offer this service and in most of the cases they also offer an API to perform query automatically. Some of these sources are listed below. The ones marked as private needs an account which can only be obtained upon invitation.

- **MalwareBazaar** (Public): contains a big collection of samples, these can be searched by their hash, family name, signature, file type etc. It offers an API which does not require an account to be queried [39].
- **VirusShare** (Private): offers a huge collection of samples, these can be searched only by their hashes, and optionally it provides useful information such as a description of the samples and how they have been classified by many AVs. Access to the API is granted only to authenticated user and the number of request per minute is limited [40].
- **VX-Underground** (Public): it is the largest collection of malware source code, samples, and papers on the internet. It was created in May, 2019 [41].
- **Malpedia** (Private): contains many resources (articles, books etc.) regarding malware. It offers an API with some public endpoints which can be used to obtain information about a specific family or threat actor and some private endpoint to perform further actions such as downloading samples, collecting YARA rules etc [42].

## 3.2 Malware execution

Despite most information can be extracted from the malware without having it running, examining the malicious software while it is executing is necessary in most cases. In fact there are lots of techniques that make static approaches too much complex to be carried on, or in some cases, completely useless. An example could be a malware that uses a C&C server to receive a malicious module that will not be available until the malware is executed. Another, less specific, example could simply be the usage of obfuscation techniques or the insertion by the malware authors of useless code that will lead to a waste of time for the analyst.
Running a malware is not something that an analyst can do on its main device without taking any precautions, but it is important to set up an analysis environment. The environment must respect some properties [43] such as:

- **Reliability**: the malware must not be capable of compromise the analysis data or gain control of the system.
- **Undetectable**: if the malware can detect the environment probably it won't perform any malicious activity making the analysis useless.
- **Isolation**: an environment not properly isolated (i.e. internet connection available, shared folders etc.) could lead to the spreading of the malicious software to other systems.

An isolated analysis environment is commonly referred as a **sandbox**.

### 3.2.1 Sandboxing

There are various practical ways to build a sandbox for dynamic analysis [43]. When talking about malware analysis, the most usual choice is a Virtual Machine. The main benefit of a VM is the strong isolation that if properly configured it guarantees and, on top of that, the possibility to easily revert the virtual system to a previous state by taking snapshots, nullifying any harmful activity done by the malware. The main disadvantage is that it requires a considerable amount of resources, especially in the case where there are multiple sandboxes running at the same time.
Once a VM is set up there are two option to perform analysis:
**- Monitoring from the VM itself.** Analysis tools are installed in the same VM where the malware runs, this approach is limited to analyze malware that operates only in the user-land.
**- Monitoring from the Host OS.** Tools required for the analysis are installed outside of the VM. The malware is tracked from the host OS. This technique is referred as *Virtual Machine Introspection* (VMI). The monitoring is performed through a component, usually a driver, installed in the VM. This approach gains the possibility to monitor also kernel actions but the injected driver can be detected by the malware.
These two layouts are illustrated in Fig. 3.1.

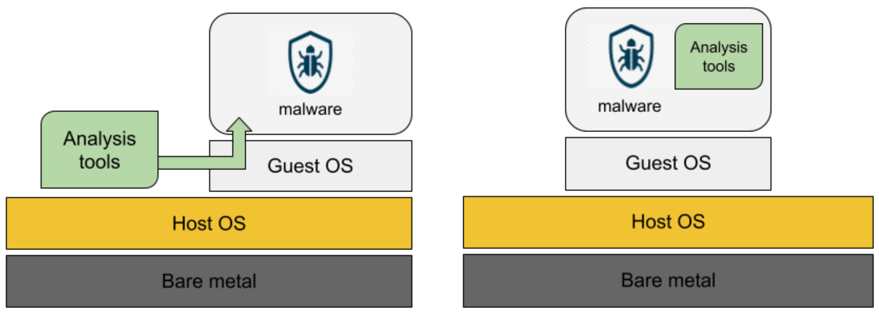


Figure 3.1. Virtual Machine layouts: VM introspection (left), Monitoring from the VM itself (right)

There are a lot of tools that are commonly used for the creation of a VM such as VirtualBox, VMWare and in the case of a Windows host there is also the Windows Sandbox [44]. Another option that is becoming more popular is the usage of *containers*. This approach in not yet widespread since it is still not clear how secure it is. Containers share the hardware with the OS so if the malware escapes the container the whole system is compromised. Agarwal et al. [45] propose a containerized environment for malware analysis. Another more popular example is REMnux [3], a linux distribution explicitly designed for malware analysis which comes with some analysis tools already installed. These tools run inside containers.

### 3.2.2 Analysis techniques and tools

There is not a standard approach to analyze malware. Each sample may come in a different format and have different layers of obfuscation, so it could require sophisticated techniques to extract useful information. Some of the most notorious techniques [46, 43] are explained in the paragraphs down below.

#### Function call analysis

Function calls can often give a good overview of the malware behavior. Despite this technique may not be really good for family classification, it often works with identification [5, 47, 48] or with type classification [49]. When it comes to analysing function calls, the main technique employed is *hooking*, which consist in a manipulation of the program behavior to intercept each function. The function hooked can be of various type: APIs, system calls or system specific APIs (i.e. Windows native API).
**Tools**. An example of programs that can be used to fulfill this task are *ltrace, strace* and *APIMonitor* [50]. The first two are available on Linux and they trace library and system calls respectively, meanwhile APImonitor works on Windows and traces Win32 APIs reporting function arguments and return values as well.

#### Controlling the execution

Taking control of the malware while it is running can prevent it from harm the system while still keeping tracks of its actions. There are many methods to achieve this task [43, 51] such as:

- **Debugging**: opcode instruction are trapped, allowing the analyst to insert breakpoints in specific places, observe the memory of the analyzed process and so on. This technique is easily detectable by malware.

- **Dynamic Binary Instrumentation (DBI)**: it consists in the insertion of instrumentation code while the malware is running. The instrumentation is usually achieved using a Just In Time (JIT) compiler. Intel PIN [52] and DynamoRIO [53] are examples of DBI frameworks. This technique is not immune to detection since it leaves traces on the system as well as debuggers and VMs do. Polino et al. [54] have developed *Arancino*, a tool that can be used to masquerade DBI traces.

Tab. 3.1 below reports some examples of tools commonly used to monitor the execution of a process with a brief description for each of them.

| Technique | Tool | Description |
|---|---|---|
| Debugging | x64dbg | Debugger for PE files [55]. |
| Debugging | WinDbg | Debugger for PE files [56]. |
| Debugging | APKtool | Android reverse engineering tool, it offers a debugger as well [57]. |
| Debugging | Android Studio | IDE for Android apps, provide a debugger also for pre-built APKs [58]. |
| DBI | Drltrace | Function call tracer based on DynamoRIO [59]. |
| DBI | Sniper | Function call tracer based on Intel PIN [60]. |

Table 3.1. Example of tools for execution control.

#### Trigger analysis

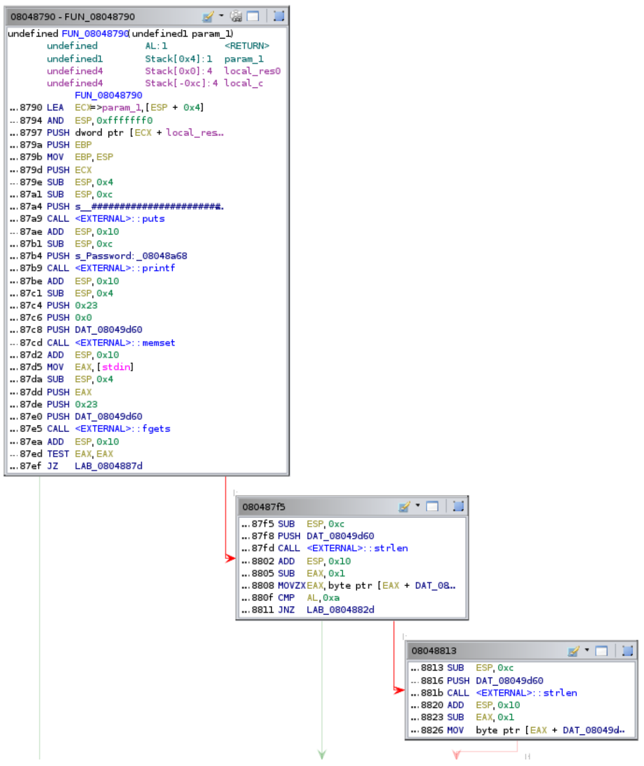
The goal is to understand what are the conditions that make the malware perform malicious activity. An example of such conditions can be listening for commands from a C&C server and not performing malicious actions until something valid is received, or checking the presence of a virtual environment to avoid detection. Having knowledge of these triggers can be very important in some situations: for example, a malware which has detected the analysis environment will result as a benign and legitimate software.
Automating this type of analysis is not an easy task, since approaches like fuzzing a malware sample can be useless. Tirenna P. [61] has developed a tool called *Symba* which uses *angr*, a symbolic analysis framework, to find triggers.

## 3.3 Manual analysis

When it comes to manually analyze a sample there are several tools that come to hand. As explained in the previous chapter in Section 2.2, there are many aspects that play an important role in malware analysis (i.e. persistence mechanisms, network communication etc.). This section will present a series of tools that are commonly adopted to investigate on those facets, reporting some usage examples as well.

### 3.3.1 Reverse engineering malware

The first approach that may come to mind is to try to get the malware code and navigate through it to understand what will be its behavior. In order to do so, there are many tools that offer disassemblers, decompilers and other useful features (i.e. CFG generator). In the context of malware analysis, IDA Pro [62] seems the most used tool, supporting a variety of file types, from simple executables to DLLs and so on. Being that famous, malware researchers have developed useful extension that may be helpful during the analysis of samples [63, 64].
Another similar tool, this time an open source one, is Ghidra [2].



```
08048790 - FUN_08048790
undefined FUN_08048790(undefined1 param_1)
        undefined         AL:1            <RETURN>
        undefined1        Stack[0x4]:1    param_1
        undefined4        Stack[0x0]:4    local_res0
        undefined4        Stack[-0xc]:4   local_c
             FUN_08048790
...8790 LEA   ECX=>param_1,[ESP + 0x4]
...8794 AND   ESP,0xfffffff0
...8797 PUSH  dword ptr [ECX + local_res..
...879a PUSH  EBP
...879b MOV   EBP,ESP
...879d PUSH  ECX
...879e SUB   ESP,0x4
...87a1 SUB   ESP,0xc
...87a4 PUSH  s__########################.
...87a9 CALL  <EXTERNAL>::puts
...87ae ADD   ESP,0x10
...87b1 SUB   ESP,0xc
...87b4 PUSH  s_Password:_08048a68
...87b9 CALL  <EXTERNAL>::printf
...87be ADD   ESP,0x10
...87c1 SUB   ESP,0x4
...87c4 PUSH  0x23
...87c6 PUSH  0x0
...87c8 PUSH  DAT_08049d60
...87cd CALL  <EXTERNAL>::memset
...87d2 ADD   ESP,0x10
...87d5 MOV   EAX,[stdin]
...87da SUB   ESP,0x4
...87dd PUSH  EAX
...87de PUSH  0x23
...87e0 PUSH  DAT_08049d60
...87e5 CALL  <EXTERNAL>::fgets
...87ea ADD   ESP,0x10
...87ed TEST  EAX,EAX
...87ef JZ    LAB_0804887d
```

```
080487f5
...87f5 SUB   ESP,0xc
...87f8 PUSH  DAT_08049d60
...87fd CALL  <EXTERNAL>::strlen
...8802 ADD   ESP,0x10
...8805 SUB   EAX,0x1
...8808 MOVZX EAX,byte ptr [EAX + DAT_08..
...880f CMP   AL,0xa
...8811 JNZ   LAB_0804882d
```

```
08048813
...8813 SUB   ESP,0xc
...8816 PUSH  DAT_08049d60
...881b CALL  <EXTERNAL>::strlen
...8820 ADD   ESP,0x10
...8823 SUB   EAX,0x1
...8826 MOV   byte ptr [EAX + DAT_08049d..
```

Figure 3.2.  An example of CFG generated by Ghidra.

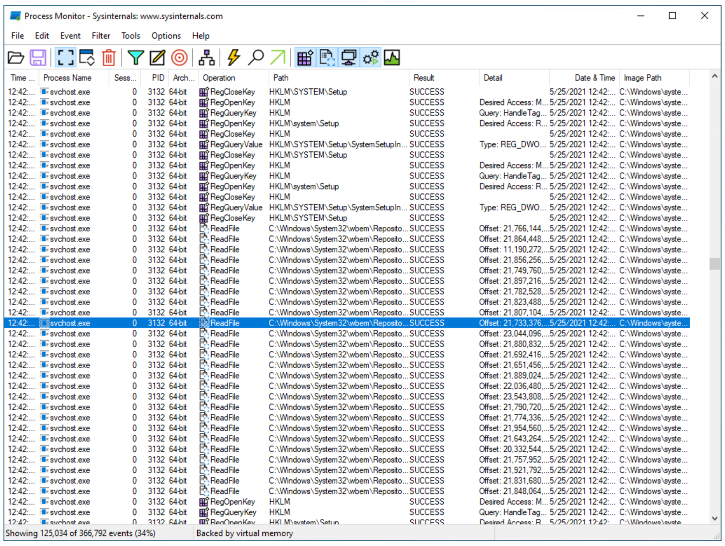
It is important to underline that understanding the complete behavior of a malware only by its decompiled/disassembled code is really hard, it could require a huge amount of time in the majority of the cases. Usually, the best option is to combine these kind of tools with other analysis softwares, which may extract automatically useful information, and then go back to the code having already partial knowledge of the malware sample. Ghidra and IDA can be used for Android applications as well, but there are tool that are more specific and designed for them:
**- Apktool**: This tool can be used to reverse engineer Android applications, it also allows to rebuild the application after it has been modified. It also offers a debugger [57].
**- dex2jar**: It can be used to work with *.dex* and *.class* files. It converts these kind of files to ASM or to smali [65].

### 3.3.2 Analysing malware with Sysinternal suite

For what concerns Windows malware there are specific tools that suite very well to perform analysis [4]. These kind of tool explicitly aim to extract specific information, such as the processes created, operations performed on the OS and so on. This troubleshooting programs where originally thought to help system administrators, but some of them have become particularly useful in the context of malware analysis :

- **Process Explorer & Process Monitor**: These two tools show running processes, like the native task manager, but with much more detailed information. Process Explorer show which handles and DLLs a process have opened. Process Monitor logs every activity done on the system by processes, allowing to filter results in order to find specific actions. This utility becomes really useful to investigate on the traces left by a sample (i.e. file dropped, registry keys written etc.).
- **Sysmon**: *System Monitor* is a system service that will persist across reboots to collect system events such as network connections, process and thread creations, drivers loading etc.

- **Autoruns**: This program adapts very well when the objective is to find persistence mechanism. It basically displays every service, process or task that will automatically run once the device is booted. It allows to insert filters to speed up the research.
- **Strings**: it can be used to extract embedded strings from a file. In some case this can give precious insight on what is being analyzed, for example in the case of a ransomware the ransom note could be found.



Process Monitor - Sysinternals: www.sysinternals.com

File Edit Event Filter Tools Options Help

| Time ... | Process Name | Sess... | PID | Arch... | Operation | Path | Result | Detail | Date & Time | Image Path |
|---|---|---|---|---|---|---|---|---|---|---|
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegCloseKey | HKLM\SYSTEM\Setup | SUCCESS | | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegOpenKey | HKLM | SUCCESS | Desired Access: M... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegQueryKey | HKLM | SUCCESS | Query: HandleTag... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegOpenKey | HKLM\system\Setup | SUCCESS | Desired Access: R... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegCloseKey | HKLM | SUCCESS | | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegQueryValue | HKLM\SYSTEM\Setup\SystemSetupIn... | SUCCESS | Type: REG_DWO... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegCloseKey | HKLM\SYSTEM\Setup | SUCCESS | | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegOpenKey | HKLM | SUCCESS | Desired Access: M... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegQueryKey | HKLM | SUCCESS | Query: HandleTag... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegOpenKey | HKLM\system\Setup | SUCCESS | Desired Access: R... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegCloseKey | HKLM | SUCCESS | | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegQueryValue | HKLM\SYSTEM\Setup\SystemSetupIn... | SUCCESS | Type: REG_DWO... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegCloseKey | HKLM\SYSTEM\Setup | SUCCESS | | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,766,144,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,864,448,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 11,190,272,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,856,256,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,749,760,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,897,216,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,782,528,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,823,488,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,807,104,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,733,376,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 23,044,096,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,880,832,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,692,416,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,651,456,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,889,024,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 22,036,480,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 23,543,808,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,790,720,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,774,336,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,954,560,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,643,264,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 20,332,544,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,757,952,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,921,792,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,831,680,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | ReadFile | C:\Windows\System32\wbem\Reposito... | SUCCESS | Offset: 21,848,064,... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegOpenKey | HKLM | SUCCESS | Desired Access: M... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegQueryKey | HKLM | SUCCESS | Query: HandleTag... | 5/25/2021 12:42:... | C:\Windows\syste... |
| 12:42:... | svchost.exe | 0 | 3132 | 64-bit | RegOpenKey | HKLM\system\Setup | SUCCESS | Desired Access: R... | 5/25/2021 12:42:... | C:\Windows\syste... |

Showing 125,034 of 366,792 events (34%) Backed by virtual memory

Figure 3.3. List of actions in Process Monitor, image taken from [66].

## 3.3.3 Dealing with APKs

APKs have a totally different structure compared to Windows applications, hence it is not possible to use the tools mentioned in Sec. 3.3.2 which are compatible only with the Windows OS. In order to deal and interact with Android applications, and more specifically, with Android malware, there are various tool that come to hand.

### Device emulation and interaction

Differently from Windows application, which can run directly on the analysis environment, or in a VM on top of it, APKs run on mobile device which has to be emulated. There are different device emulators that are freely available such as Anbox and Android-86. Once an emulator has been properly set up the analyst needs to interact with it. It can be done by using *Android Debug Bridge (adb)* [67] that is a command line utility through which is possible to install debug apps and have a Unix shell to execute commands.
Once a device is being emulated an analyst could want to perform actions that are similar to the ones a real user would perform. *MonkeyRunner* is a tool which lets us to perform this kind of actions. It permits to send keystrokes, take screenshots, simulate clicks etc. from outside the device. Below is reported an usage example, taken and re-elaborated from [68]:

Listing 3.1. Monkeyrunner script example

```
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice
device = MonkeyRunner.waitForConnection()
package = 'com.example.android.maliciousapp'
activity = 'com.example.android.maliciousapp.MainActivity'

# sets the name of the component to start
```

```
runComponent = package + '/' + activity
# Runs the component
device.startActivity(component=runComponent)
# Presses the Menu button
device.press('KEYCODE_MENU', MonkeyDevice.DOWN_AND_UP)
# Takes a screenshot
result = device.takeSnapshot()
# Writes the screenshot to a file
result.writeToFile('myproject/shot1.png','png')
```

## 3.4 Automated analysis

Even if manual analysis is in many cases necessary (i.e. to perform forensics investigation after a malware attack) it consumes a considerable amount of time and apply "manual" techniques to a large number of samples is unfeasible, so it becomes necessary to employ an automated approach. This section will dive into the usage of tools that automates the analysis process, running a sample in a secure environment and generating a report which describes the malware activities.

### 3.4.1 Cuckoo Sandbox

The Cuckoo sandbox [69] is an open source tool employable for the analysis of software. It basically uses a VM as a sandbox where suspicious files are executed and collects any useful information that has been generated during execution. There are various works that have used the Cuckoo Sandbox [70, 71, 6]. This tool is able to perform various actions such as:

- Analyze different types of files: executables, DLLs, office documents, Android APKs etc.
- Trace API calls: returns a list of all the API called as well as their return value.
- Dump network traffic: network packets can be captured and analyzed.
- Memory analysis: uses Volatility combined with YARA rules.

#### Architecture

Cuckoo is made of at least 3 basic components: a central host, an analysis environment and a virtual network. The central host is the one in charge of submitting samples to the analysis environment and generate reports. The environment (or guest) is the actual sandbox where the samples will be ran, here a cuckoo agent runs and it is the component that communicates with the host. The virtual network is an isolated link through which the the host and the guest communicates. Cuckoo grants the possibility to launch multiple analysis in parallel by running multiple guests at the same time. Each guest may have its own configuration. Network setting can be customized as well, these are the available options:

- **None**: host is not connected to the internet, so there will be no packet captured.
- **Drop**: only the cuckoo messages are allowed, anything else is just dropped.
- **Internet**: guest has full access to the Internet, this option is the most dangerous.
- **InetSim**: it uses InetSim, a network utilities emulator, that will return custom responses when the guest tries to access Internet.
- **TOR**: traffic is routed through TOR.
- **VPN**: traffic is routed through one or multiple VPN endpoints.

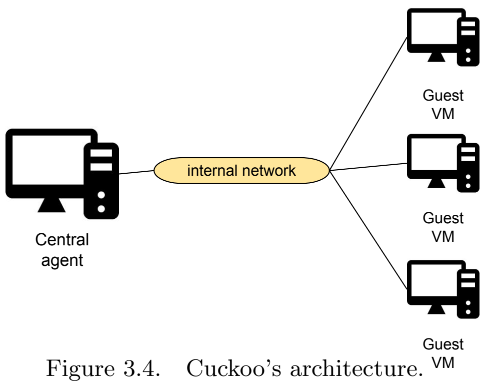
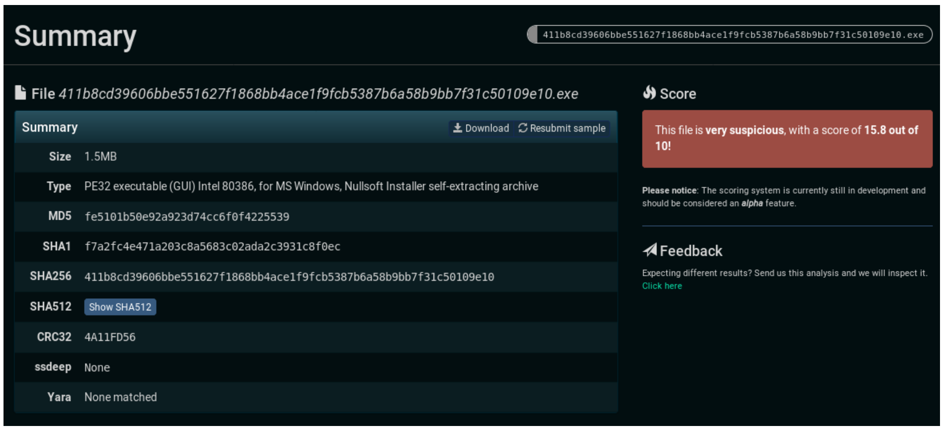


Figure 3.4. Cuckoo's architecture.

On top of these components Cuckoo offers various utilities. Some of them are described below.
**Web interface**. It is based on a Mongo database, it allows to submit files, analyze reports and view previous analysis.
**Submission utility**. This is used to submit a sample to cuckoo, can be used through the Web interface, the API or the CLI.
**Community utility**. It downloads signatures that can be later used to automatically classify files.

### Reports

After an analysis is successfully completed, a report will be generated. This report will contain various kind of information, that differs based on what the malware has performed and how the whole sandbox was configured. Figures 3.5 and 3.6 below show an example of such reports.



Summary

411b8cd39606bbe551627f1868bb4ace1f9fcb5387b6a58b9bb7f31c50109e10.exe

File *411b8cd39606bbe551627f1868bb4ace1f9fcb5387b6a58b9bb7f31c50109e10.exe*

Summary — Download — Resubmit sample

| | |
|---|---|
| **Size** | 1.5MB |
| **Type** | PE32 executable (GUI) Intel 80386, for MS Windows, Nullsoft Installer self-extracting archive |
| **MD5** | fe5101b50e92a923d74cc6f0f4225539 |
| **SHA1** | f7a2fc4e471a203c8a5683c02ada2c3931c8f0ec |
| **SHA256** | 411b8cd39606bbe551627f1868bb4ace1f9fcb5387b6a58b9bb7f31c50109e10 |
| **SHA512** | Show SHA512 |
| **CRC32** | 4A11FD56 |
| **ssdeep** | None |
| **Yara** | None matched |

Score

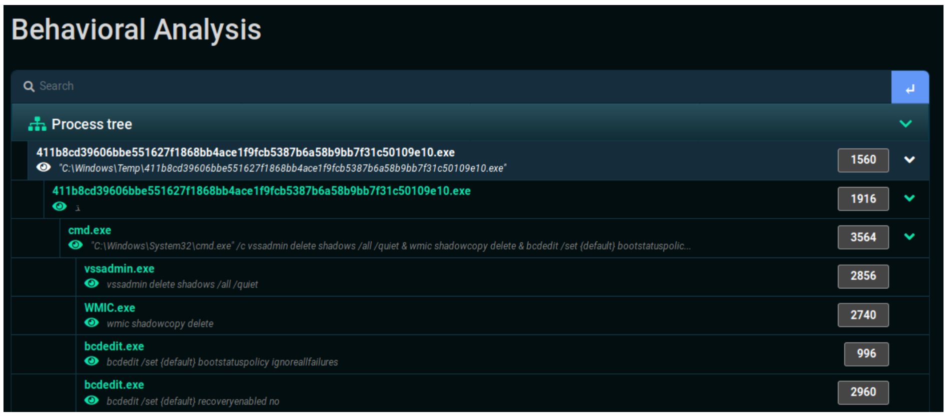
This file is **very suspicious**, with a score of **15.8 out of 10!**

**Please notice**: The scoring system is currently still in development and should be considered an ***alpha*** feature.

Feedback

Expecting different results? Send us this analysis and we will inspect it.
Click here

Figure 3.5. Cuckoo analysis summary.



```
Behavioral Analysis

Search

Process tree
411b8cd39606bbe551627f1868bb4ace1f9fcb5387b6a58b9bb7f31c50109e10.exe   1560
  "C:\Windows\Temp\411b8cd39606bbe551627f1868bb4ace1f9fcb5387b6a58b9bb7f31c50109e10.exe"
  411b8cd39606bbe551627f1868bb4ace1f9fcb5387b6a58b9bb7f31c50109e10.exe   1916
    1
    cmd.exe   3564
      "C:\Windows\System32\cmd.exe" /c vssadmin delete shadows /all /quiet & wmic shadowcopy delete & bcdedit /set {default} bootstatuspolic...
      vssadmin.exe   2856
        vssadmin delete shadows /all /quiet
      WMIC.exe   2740
        wmic shadowcopy delete
      bcdedit.exe   996
        bcdedit /set {default} bootstatuspolicy ignoreallfailures
      bcdedit.exe   2960
        bcdedit /set {default} recoveryenabled no
```

Figure 3.6. Cuckoo behavioral analysis results.

### Cuckoo Droid

It is an extension of Cuckoo to handle analysis of Android applications [72]. Differently from before, now the guest runs a Linux OS with an android emulator within. Different tools are installed both on the guest OS and the emulator to perform the analysis, these are reported in Tab. 3.2. Fig. 3.7 shows a representation of Cuckoo Droid's architecture.

| Tool | Guest OS/Emulator | Description |
|---|---|---|
| AAPT | Guest OS | Extracts the main activity and the package name from the APK. |
| ADB | Guest OS | Bridge to communicate with the emulator. |
| Xposed | Emulator | Framework to change the behavior of system and apps. |
| Droidmon | Emulator | API call monitor. |
| Superuser | Emulator | App that grants Superuser rights. |
| Content generator | Emulator | Generates content on the device to make it more realistic. |
| Emulator anti-detection | Emulator | Collection of anti-detection techniques. |

Table 3.2. Cuckoo Droid requirements.

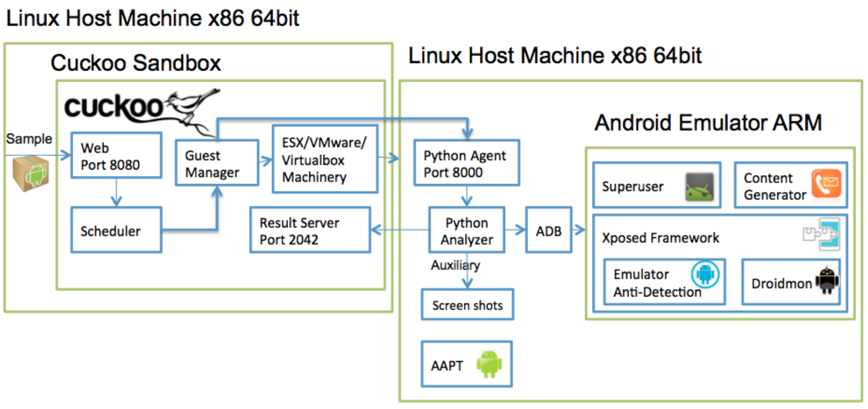


Figure 3.7. Cuckoo droid architecture, image taken from its documentation [72].

As in the standard Cuckoo sandbox there is a web interface to provide easier interaction, where it is possible to submit samples and customise the analysis. Once an analysis is completed a report will be generated, containing information about what happened during execution as well as some static information (i.e. permission the app requires).

### 3.4.2 Online analysis

Another way to automate the analysis process is by using online sandboxes. These are of course the simplest to use, since they do not require any kind of setup or resources to run the analysis. So from the analyst's point of view the whole analysis is reduced to a simple upload of the sample and the analysis of a generated report, having, in most cases, no idea of what is happening beyond. The main advantage of this approach is undoubtedly the avoidance of having enough resources to setup an environment. On the other side, the main disadvantage is that these services are often expensive, being suited for large companies or professional analysts. Generally they offer a free version which raises limits in the number of analysis or requests available.
There are various options to perform analysis by the means of online sandboxes, these usually offer similar functionalities: they provide an environment where the malware will be launched and then an analysis report which describes what is happened during the execution will be generated. Depending on the sandbox the environment may be more or less customized (i.e. selecting a specific OS, a preferred network configuration etc.). The reports generated by these sandbox are usually very similar as well, containing a trace of the API called, the operation performed on the system, network activity etc.
Some examples of such sandboxes are *Any.Run* [73] and *Triage* [74].

#### Any.Run

It is an analysis platform, it offers some services for free (but with limitations). Basically it provides a Windows sandbox where the malware will be injected and run for at most five minutes. After that a report is generated, this will contain: static information (i.e. number of sections, imports etc.), a process graph, network and files activities, operations performed on the Windows registry and some screenshots. There is also the possibility to download a file containing the sample's Indicators Of Compromise (IOCs), the network packets capture and the attack matrix which is a report of known attack used by the sample. Figures 3.8 and 3.9 below report an example of the process graph and the attack matrix for a sample belonging to the Trickbot family.

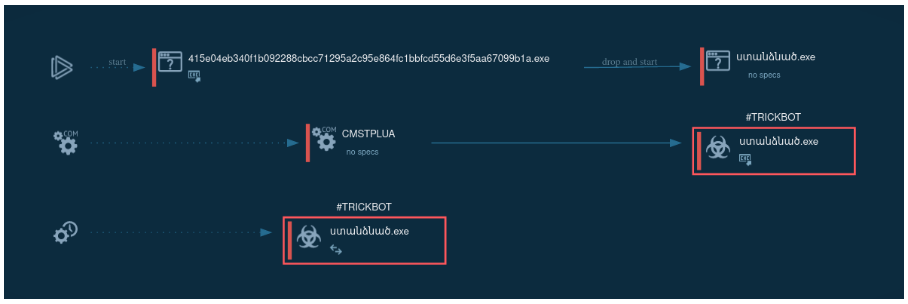
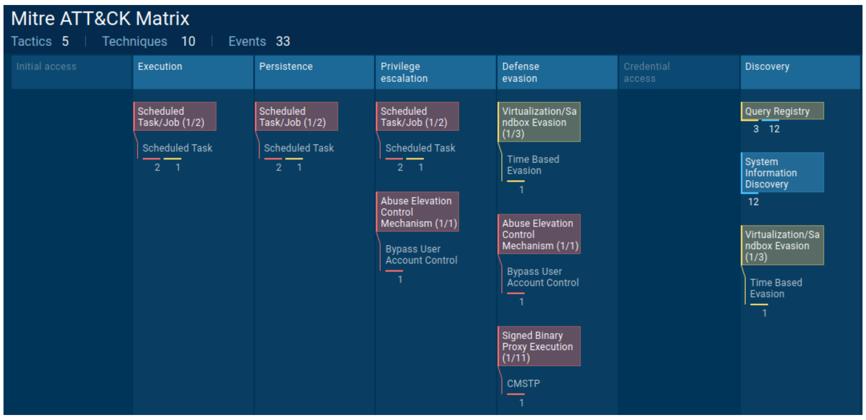


Figure 3.8. Process graph.



Mitre ATT&CK Matrix
Tactics 5 | Techniques 10 | Events 33

Initial access

Execution
Scheduled Task/Job (1/2)
Scheduled Task
2 1

Persistence
Scheduled Task/Job (1/2)
Scheduled Task
2 1

Privilege escalation
Scheduled Task/Job (1/2)
Scheduled Task
2 1
Abuse Elevation Control Mechanism (1/1)
Bypass User Account Control
1

Defense evasion
Virtualization/Sandbox Evasion (1/3)
Time Based Evasion
1
Abuse Elevation Control Mechanism (1/1)
Bypass User Account Control
1
Signed Binary Proxy Execution (1/11)
CMSTP
1

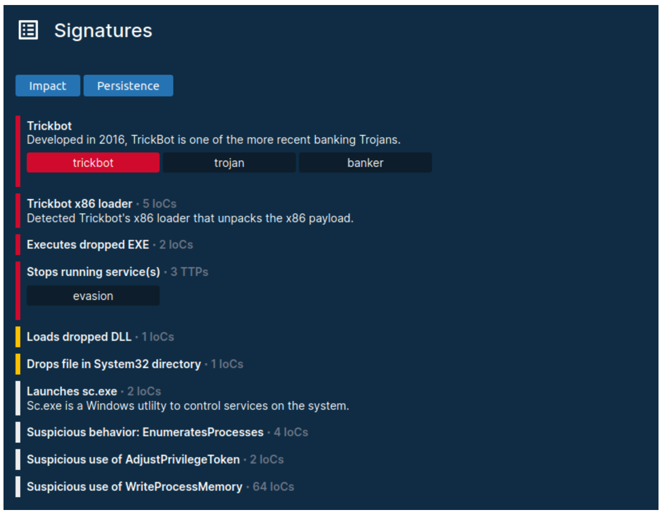
Credential access

Discovery
Query Registry
3 12
System Information Discovery
12
Virtualization/Sandbox Evasion (1/3)
Time Based Evasion
1

Figure 3.9. Attack matrix.

### Tria.ge

It is another popular sandbox, it allows to submit different types of files and customize the execution environment, from the OS version to the Internet connection and the analysis duration. Once the analysis is started, there is also the possibility to extend is duration and interact with the machine. After that, a report will be generated which will contain insights on the malware activities, such as file and registry operations, process created, network traffic and so on. It allows also to download memory dumps and network captures (i.e. pcap files). It also generate an attack matrix.
By upgrading to a researcher account it is also possible to use the sandbox programmatically thanks to its API, by which the analyst can submit samples and retrieve reports and generated files. The Fig. 3.10 below shows an example of a report for a sample of the Trickbot family.



Signatures

Impact Persistence

**Trickbot**
Developed in 2016, TrickBot is one of the more recent banking Trojans.
trickbot trojan banker

**Trickbot x86 loader** · 5 IoCs
Detected Trickbot's x86 loader that unpacks the x86 payload.

**Executes dropped EXE** · 2 IoCs

**Stops running service(s)** · 3 TTPs
evasion

**Loads dropped DLL** · 1 IoCs

**Drops file in System32 directory** · 1 IoCs

**Launches sc.exe** · 2 IoCs
Sc.exe is a Windows utilty to control services on the system.

**Suspicious behavior: EnumeratesProcesses** · 4 IoCs

**Suspicious use of AdjustPrivilegeToken** · 2 IoCs

**Suspicious use of WriteProcessMemory** · 64 IoCs

Figure 3.10. Triage report for a Trickbot sample.

## 3.5 Extracting features

During the analysis process the relevant information can be extracted not only to be further analyzed but also to become input to a machine learning system, which will be trained to automate the classification process. These features can be of many kinds, from simple strings to actual pieces of code. Generally speaking, on the basis of the analysis that has been performed, there are two kinds of features: *static* and *dynamic*. Tables 3.3 and 3.4 report some examples of such features.

| **Feature** | **Description** |
|---|---|
| Strings | All printable strings contained in a sample, can contain URLs, paths etc. |
| API calls | Certain API calls are index of suspicious activity, and the usage of some of them can lead to a proper classification. |
| OPcode | Code bytes can be collected in sequences called *n-grams*, useful since there is some code reuse between samples of the same family. |

Table 3.3. Example of static features.

| **Feature** | **Description** |
|---|---|
| Registry operations | Registry is often modified to achieve persistence or avoid detection. |
| Network operations | Malware often tries to interact with a C&C server, hence the packets transmitted and the list of IP contacted can be a good clue of the family it belongs to. |
| File operations | Malware often open specific files or copies itself in other location and with specific names, these behavior are often preserved within a family |

Table 3.4. Example of dynamic features.

# Chapter 4

# Building a malware collection

The first step in our research was to obtain knowledge about malware families, trying to understand what are the peculiarities that define a family and which ones are more common nowadays. After acquiring this information, the next step was to find how to get malware samples. Ultimately, all these sources were combined to produce our own collection containing only those families which are more widespread today, if samples are available. This chapter will describe which are the selected families, explaining what are the reasons that led to such selection and which are the malware sources used to collect the desired samples, reporting all the difficulties encountered during this process as well.

## 4.1 Family selection

Before talking about the families that were selected, it is important to note that malware authors generally develop samples targeting one precise operating system so it was necessary to select which OS should be considered. Among all the possible candidates, statistics show that **Windows** and **Android** are both the most common and the most targeted by malware. Figures 4.1, 4.2 report these statistics.

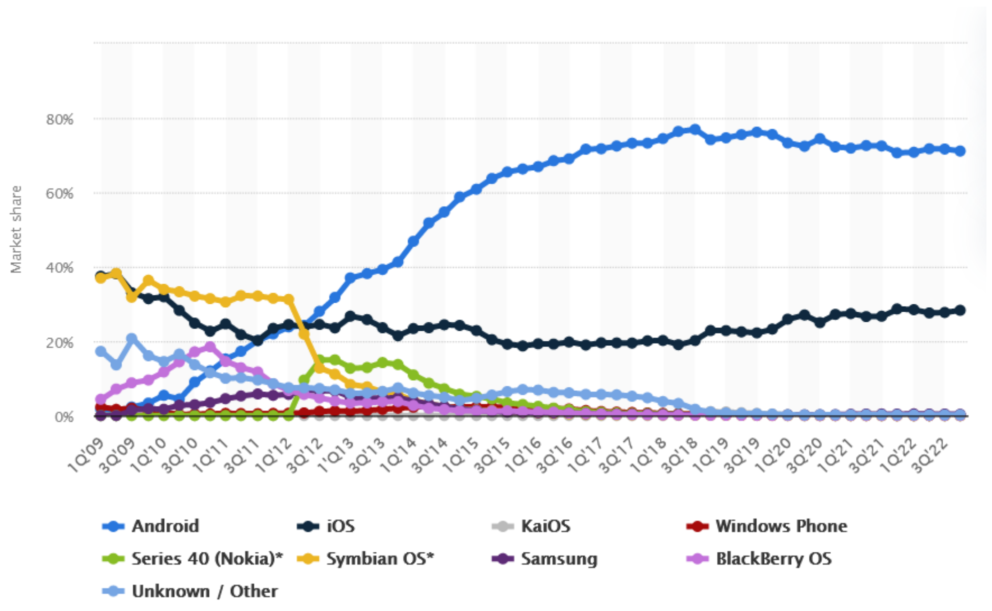


Figure 4.1. Mobile OS market share worldwide, image taken from Statista [75].

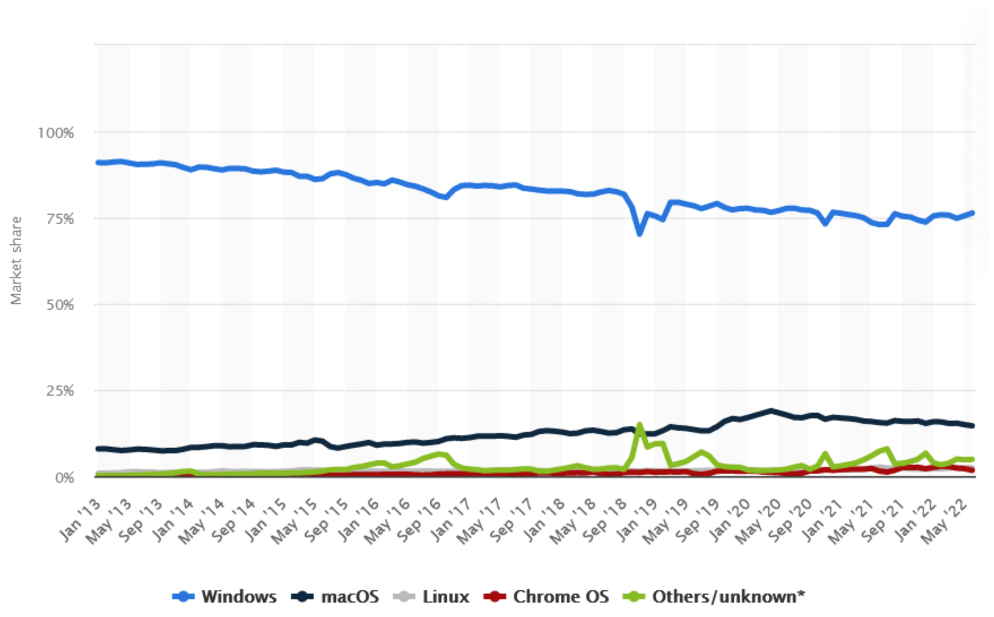


Figure 4.2. Desktop OS market share worldwide, image taken from Statista [76].

### 4.1.1 Types

Once the operating systems were selected we tried to find out if there are particular types which are more common or more dangerous than others. As already explained in Sec. 2.1.2 malware can be categorized into different types depending on what its purpose is and how it is achieved. Antivirus companies such as Kaspersky and MalwareBytes periodically publish reports [77, 78] that describe the current landscape. Tab. 4.1.1 reports some of the statistics extracted from these reports.

| **Windows** | | **Android** | |
|---|---|---|---|
| **Malware type** | **Share** | **Malware type** | **Share** |
| Trojan | 20% | Adware | 42% |
| Adware | 13% | RiskTool | 35% |
| Riskware Tool | 12% | Trojan | 15% |

Table 4.1. Malware types statistics for Android and Windows OSs.

Another important aspect that was taken into consideration to perform a type selection, was how dangerous some types can be for some targets. This led to the inclusion of another typology: *ransomware*. This category doesn't appear to be as common as the others, probably because ransomware groups are moving their interests into attacking entire organizations rather than single users [78]. Figures 4.3, 4.4 show the percentage of organizations that have been hit by ransomware in the last year. As can be evinced from the chart, more than half of each category of companies have suffered from a ransomware attack. With all this information in mind, we decided to put our focus on the following types:

- **Windows OS** : *Trojan* and *Ransomware*
- **Android OS** : *Trojan*, *Risktool* and *Adware*

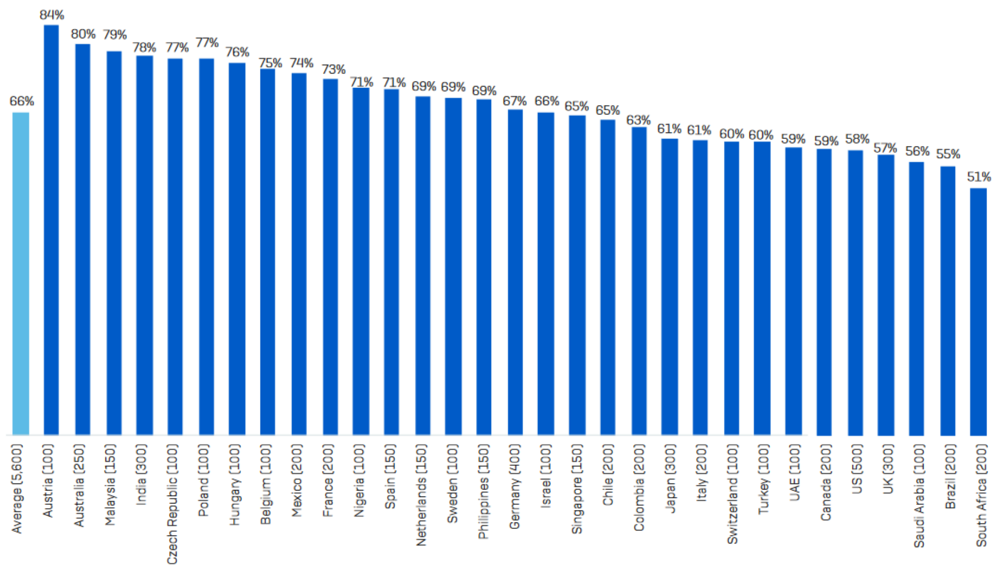


Figure 4.3. Percentage of organization hit by ransomware, by country. Image taken from Sophos [79].

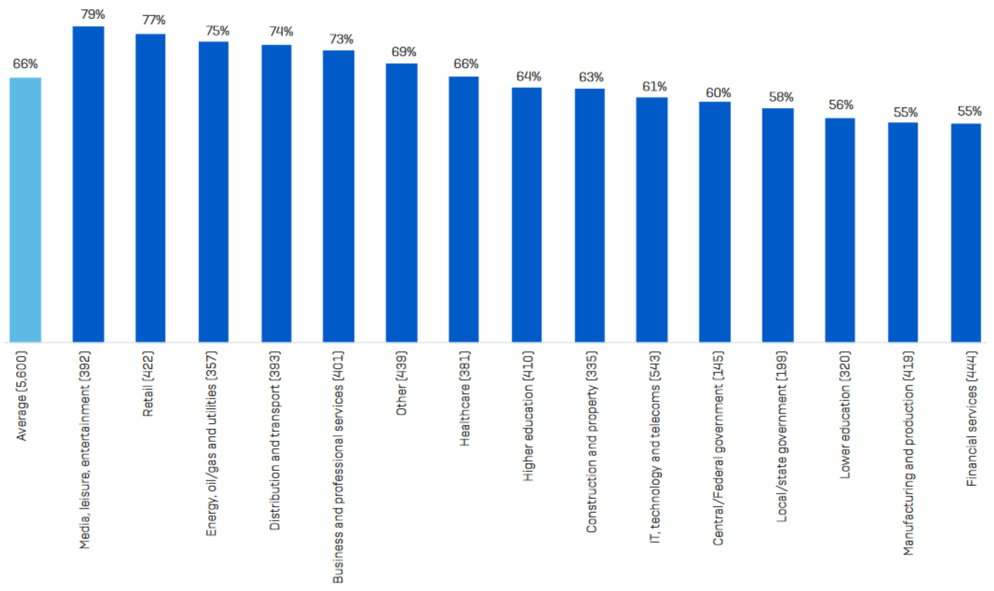
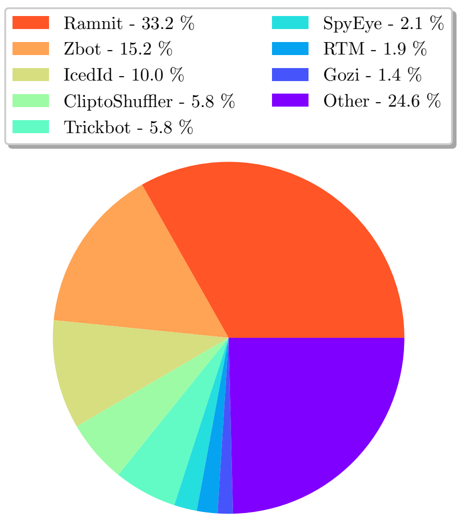


Figure 4.4. Percentage of organization hit by ransomware, by type. Image taken from Sophos [79].

### 4.1.2 Families

In order to perform a family selection we examined some of the Kaspersky quarterly reports. These reports describe the current threat landscape, reporting the number of malware detected and the family they belong to. Figures 4.5, 4.6 illustrate the top families for the selected types on both desktop and mobile systems.



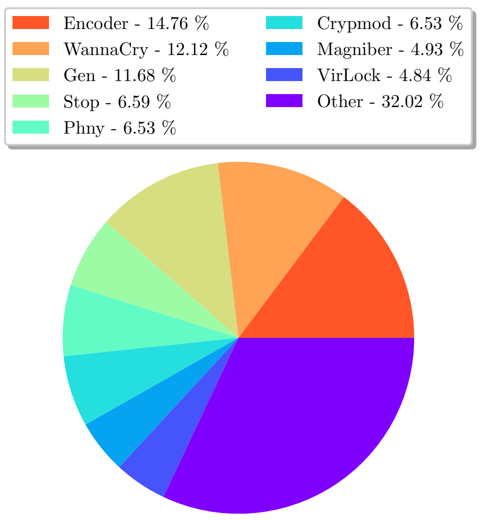
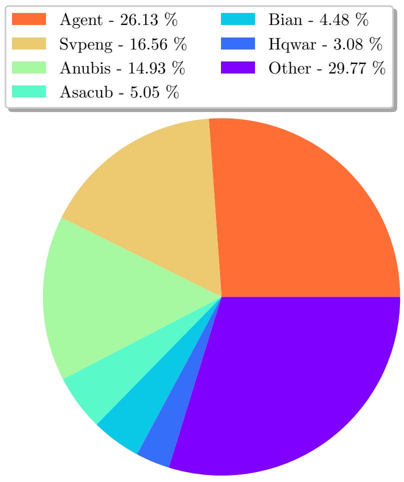
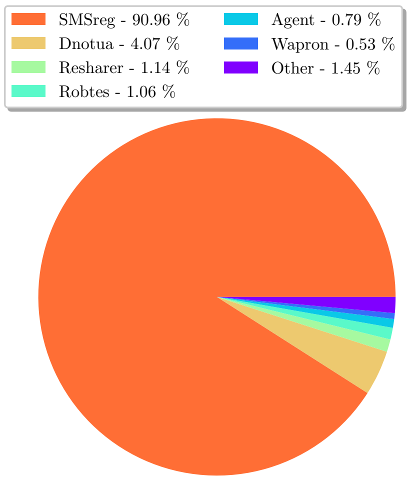


Figure 4.5. Top families for trojan and ransomware on Windows.





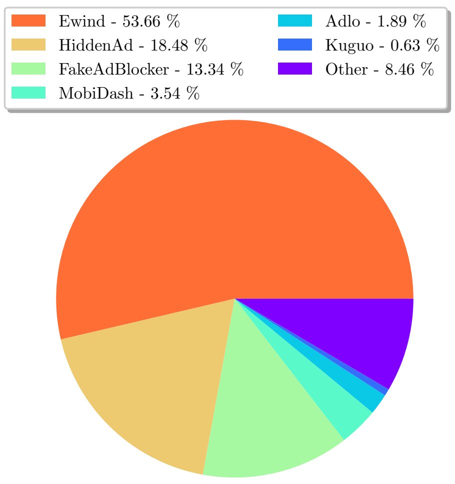


Figure 4.6. Top families for trojan, risktool and adware on Android.

These statistics have been used to decide which families we should focus on, in order to create a collection that would contain samples that are not “out of date” and still represent a serious menace today. With that being said, there is another important point that cannot be overlooked: the *availability* of samples for those families. In fact, as it will be explained in the next section, despite our desire was to include in our collection the most widespread families, this was not possible for some of them, mainly the Android ones, given the lack of a consistent number of samples on the web.

## 4.2 Collecting samples

The next step, after having depicted an initial idea of what the collection should contain, was to find reliable sources of malware samples and collect them. This resulted to be not an easy task mainly for two reasons:

- Samples availability: since for some of the families reported in Tabs. 4.5, 4.6 there were no samples available it was not possible to include them in our collection.
- Family naming: malware naming is not a well defined process, there is not a common standard. Back in the 1991 Caro attempted to create a naming convection [80] which has become outdated though. Today almost each Anti-virus company uses its own convection when performing identification of a sample and other than that there are various families which are known by multiple names. Tab. 4.2 reports some examples of families known by multiple names.

| **Family main name** | **Alternative names** |
|---|---|
| Zeus | Zbot, Wsnpoem, Citadel |
| Trickbot | Trickster, Trick, TrickB |
| RTM | Redaman |
| Ramnit | Nimnul |
| REvil | Sodinokibi, Sodin |
| PolyRansom | Virlock |
| Anubis | Bankbot, Bankspy |
| Gozi | Papras, Snifula, Ursnif, CRM |

Table 4.2. Some examples of malware alternative family names.

### 4.2.1 Malware sources

There are many malware collections available online, some of them require to have an account which can be obtained only upon invitation. In order to overcome the naming problem we used Malpedia [42] which offers a family name search engine which will return a list of technical articles and reports about the specified family, along with its alternative names and, if registered, some samples as well.

#### VirusShare

Among all the collections available the most rich and valuable is VirusShare [40], which right now contains more than 56,000,000 samples. The archive is organized in files containing the hashes of the available samples. Then it is possible to use the API to perform queries about specific samples and to download them. The main complication we had when parsing this huge archive was that the available samples are specified only by their hash, so for each of them we should have performed an initial request to obtain information about that sample (i.e. how AVs have identified it) and then download it only if it belongs to one of our selected families. Since the number of request is limited to 4 per minute, performing this task for the whole archive was unfeasible. Luckily we managed to get around this, thanks to VX-underground [41] which contains a collection of text files, named *VirusShare IOC Listings*, where for each VirusShare's sample the relative family is given. Hence we parsed these files instead of the VirusShare's ones.

#### MalwareBazaar

Another famous malware database is MalwareBazaar [39], which does not contain a collection as big as VirusShare's one (613,363 samples) but offers the possibility to perform queries directly by using family names. To enrich our collection we queried also MB and merged their samples with the ones obtained from VirusShare.

#### Datasets

In addition to the two famous malware collections mentioned above, we took samples from some already made datasets. The reason that led to the usage of these datasets is that, especially for Android, no samples were found for some of the families mentioned in Tabs. 4.5, 4.6. The dataset used are:

- **MalRadar** [81]: this dataset contains 4,534 Android malware samples scattered across 121 families. Samples have been collected by crawling the mobile security reports published by ten leading security companies and samples have been manually verified to ensure reliability.
- **Argus APK Collection**: unfortunately the main webpage where this dataset was hosted is no longer available but the entire collection can be found on VX-underground [82]. It contains a total of 24,553 Android samples divided into 71 families.
- **CIC-AndMal2017** [83]: it contains both benign and malicious Android applications. The number of malware samples is 426, categorized into 42 families which belong to four types: Adware, Ransomware, Scareware and SMS malware.
- **DikeDataset** [84]: it is a labeled dataset which contains both benign and maliciuous PE and OLE files. The dataset has been built to perform identification (i.e. being able to assert wheter a given file is malicious or benign).

### 4.2.2 How to collect samples

The creation of a large collection which shall contain thousands of samples is not something that could be done by hand. In order to fulfill this task programmatically we used, when available, the API services offered by the malware collections mentioned above. By using these APIs is possible to query a specific hash in order to obtain information about the relative sample or, in the case of MalwareBazaar, to use the family name to get all the available samples. In order to communicate easily with VirusShare's API they also provide a Python client that can be installed by running:

```
pip install PyVirusShare
```

In order to use this client the user should have a valid API key. The client gives the possibility to query a specific hash, either to check if VirusShare has it or to get some information or to download it. Listing 4.1 shows how to query a sample, the result will be a JSON which contains: the scan results from several AVs showing how that sample has been classified and some information about the file such as the type, entry point, timestamps, hashes and so on. An example of such report is shown in Listings 4.2 and 4.3.

```
from virusshare import VirusShare
v = VirusShare("apikey")
#result will contain the JSON report
result = v.info('a1ac533baaf7de1dae53cf5b465aeca28a7f20bdfc79e5a0a39437dd728c231f')

#If we wish to download the sample
v.download('a1ac533baaf7de1dae53cf5b465aeca28a7f20bdfc79e5a0a39437dd728c231f')
```

Listing 4.1. VirusShare client usage example.

```
    "scans": {
      "TotalDefense": {
        "result": "Win32/Zbot.HII"
      },
      "CAT-QuickHeal": {
        "result": "TrojanPWS.Zbot"
      },
      "AhnLab-V3": {
        "result": "Trojan/Win32.Zbot"
      },
```

Listing 4.2. Section of a VirusShare's JSON JSON reporting how the sample has been classified by AVs.

```
  "exif": {
    "CharacterSet": "Unicode",
    "CodeSize": 29696,
    "CompanyName": "Piececoast Corporation.",
    "EntryPoint": "0x2ca9",
    "FileDescription": "Piececoast Locate",
    "FileFlags": "(none)",
    "FileFlagsMask": "0x0000",
    "FileOS": "Win32",
    "FileSize": "446 kB",
    "FileSubtype": 0,
    "FileType": "Win32 EXE",
    "FileTypeExtension": "exe",
    "FileVersionNumber": "6.2.49985.2",
    "ImageVersion": 0,
    "InitializedDataSize": 577536,
    "InternalName": "Seat.exe",
    "LanguageCode": "English (U.S.)",
```

Listing 4.3. Section of a VirusShare's JSON reporting some file information.

In order to interact with MalwareBazaar API we didn't used any already made tool or library since MB offers many more endpoints compared to VirusShare and we just needed a couple of them. This endpoints can be contacted through GET/POST request. Listing 4.4 shows the python code used to contact MB and perform two actions: retrieve information for a specific hash and download the relative sample. Listings 4.5, 4.6 report some sections of the JSON returned upon the first request.

```python
import requests
MB_api = "https://mb-api.abuse.ch/api/v1/"
data = {
    "query": "get_info",
    "hash": "e7456c57dba442a7e63f2bd45ff5be6c8168f2fcfd15c5e405536fb3bb212dcb"
}
#issue a post request and retrieve the json contained in the response
res = requests.post(MB_api, data=data, timeout=10, allow_redirects=True).json()
#check if the sample was found on MB
if res['query_status'] == "ok":
    #print out all the info
    print(res['data'])

#If we wish to download a specific sample
data = {
    "query": "get_file",
    "sha256_hash": "e7456c57dba442a7e63f2bd45ff5be6c8168f2fcfd15c5e405536fb3bb212dcb"
}
res = requests.post(MB_api, data=data, timeout=10, allow_redirects=True).json()
#now check if the file was found
if res["query_status"] != "file_not_found":
    print("Sample successfully downloaded")
```

Listing 4.4. Code used to contact MB endpoints.

```json
"data": [
  {
    "sha256_hash": "e7456c57dba442a7e63f2bd45ff5be6c8168f2fcfd15c5e405536fb3bb212dcb",
    "sha3_384_hash": "ec210365723b02ba6fedcd33ca6ca0b67a0d18d9045a322af81a57bf9980c41b15e898b30dfa462a4eb7ef3089463b06",
    "sha1_hash": "a2ce7a730e96bf6c8f9cd512993fd67cf0c10767",
    "md5_hash": "e8b61b099af93918a7d59477334471e0",
    "first_seen": "2023-01-10 09:00:40",
    "last_seen": null,
    "file_name": "49136 E2K 610622871149136E2K 6106228711.exe",
    "file_size": 823691,
    "file_type_mime": "application/x-dosexec",
    "file_type": "exe",
    "reporter": "cocaman",
    "origin_country": null,
    "anonymous": 0,
    "signature": "TrickBot",
```

Listing 4.5. Section of a MalwareBazaar's JSON reporting some file information.

```json
"data": [
  {
    "sha256_hash": "e7456c57dba442a7e63f2bd45ff5be6c8168f2fcfd15c5e405536fb3bb212dcb",
    "sha3_384_hash": "ec210365723b02ba6fedcd33ca6ca0b67a0d18d9045a322af81a57bf9980c41b15e898b30dfa462a4eb7ef3089463b06",
    "sha1_hash": "a2ce7a730e96bf6c8f9cd512993fd67cf0c10767",
    "md5_hash": "e8b61b099af93918a7d59477334471e0",
    "first_seen": "2023-01-10 09:00:40",
    "last_seen": null,
    "file_name": "49136 E2K 610622871149136E2K 6106228711.exe",
    "file_size": 823691,
    "file_type_mime": "application/x-dosexec",
    "file_type": "exe",
    "reporter": "cocaman",
    "origin_country": null,
    "anonymous": 0,
    "signature": "TrickBot",
```

Listing 4.6. Section of a MalwareBazaar's JSON reporting AVs classification.

### 4.2.3 Collecting Windows samples

For what concerns Windows malware the main source of samples used have been VirusShare and MalwareBazaar. The first step was to collect only hashes for valid samples. A Windows sample is *valid* if it meets two conditions:

- It belongs to one of the selected families.
- It is a Portable Executable (PE).

The second condition, which may sound a little restrictive, was chosen to simplify the analysis process avoiding to handle different formats such as dlls, office documents and so on. As already mentioned above in Sec. 4.2.1, we used the *VirusShare IOC Listings* collection from VX-underground to have a prior classification of VS samples. We downloaded every file from this collection, parsed each of them and when a sample belonging to our families was found we contacted VirusShare to check the file type (i.e. if it is a PE). At first we just did an enumeration of samples, in order to have a complete view of how many samples were available for each family. We started considering all the families reported in Kaspersky's reports, cutting out the ones for which no samples, or too few, were available. We performed the same query on MalwareBazaar and merged the results. Tab. 4.2.3 shows a list of families for which at least 100 samples was found.

| **Family name** | **Number of available samples** | **Type** |
|---|---|---|
| Ramnit | 2528 | Trojan |
| Zbot | 2827 | Trojan |
| IcedId | 1672 | Trojan |
| Trickbot | 1100 | Trojan |
| SpyEye | 2505 | Trojan |
| Danabot | 1083 | Trojan |
| Tinba | 571 | Trojan |
| Gozi | 2863 | Trojan |
| Stop | 1477 | Ransomware |
| Virlock | 2520 | Ransomware |

Table 4.3. Number of samples available for some of the selected Windows families.

As it can be inferred by this list, there are just two Ransomware families for which a consistent number of samples is available. Since our wish was to have this dataset balanced both in the number of samples per family and in the number of samples per type, it was necessary to look for other sources of samples. We found a considerable amount of samples for *Wannacry* and *Magniber* families on VX-underground in their *In The Wild Collection*, so we extracted them manually and added to our initial collection [85].

### 4.2.4 Collecting Android samples

Finding Android samples was not as easy as for Windows' ones. We enforced two conditions to retain a sample as valid:

- It belongs to one of the selected families.
- It is an Android Package (APK).

We could have also considered *.dex* files as well, but APKs contain more information about the application so it would probably be better for analysis. Other than just VirusShare and MalwareBazaar, some dataset have been partially used since they seemed to be the only available sources for some families. Obviously these dataset didn't offer any kind of API, each of them specifies the contained samples in its own way, hence the collection process could not be automatized as in the previous case and we needed to download the whole datasets and extract only what was needed. Tab. 4.4 reports the number of samples found for a subset of the selected families.

## 4.3 Putting all together

Once all the samples available for the families were enumerated, we were ready to proceeded with the actual creation of the collection. Before doing that, it was necessary to define how big this collection should be and how many families it should contain. To this regard, we took the following considerations:

| Family name | Number of available samples | Type | Source used |
|---|---|---|---|
| Ewind | 10 | Adware | CICAndMalDataset |
| MobiDash | 10 | Adware | CICAndMalDataset |
| HiddenAd | 288 | Adware | MalRadar Dataset |
| Kuguo | 100 | Adware | Argus Collection |
| Hqwar | 179 | Trojan | VirusShare & MalwareBazaar |
| Anubis | 46 | Trojan | VirusShare & MalwareBazaar |
| Svpeng | 23 | Trojan | VirusShare & MalwareBazaar |
| Asacub | 29 | Trojan | VirusShare & MalwareBazaar |

Table 4.4. Number of samples available for some of the selected Android families.

- First of all, we decided to separate Microsoft samples from Android ones. The main reason is that they are completely different file types, hence the kind of features that will be extracted will be different as well.
- Since the number of available Android's samples was definitely smaller than Microsoft's, we decided to build the Android collection with the only purpose to be analyzed by using different techniques (i.e. static and/or dynamic), meanwhile the Microsoft collection will be a *dataset* that can also be used in a Machine Learning system to perform both identification and characterization.
- For what concerns the *dataset* the number of samples of different classes should be balanced since it will affect the Machine Learning precision. Furthermore, it would be even better if the number of samples reflects what the reality actually is, which basically means: the more common a family is the more samples the dasaset shall have. This affects also the number of classes since we had to discard those families for which there were not enough sample in order to preserve this balance.

### 4.3.1 ST-WinMal dataset

The ST-WinMal dataset contains a total of 8 classes of Windows malware (i.e. families) plus another class of only benign files which has been inserted in order to perform both characterization (distinguish a certain family from another one) and identification (defining a sample as malicious or benign). The total number of sample for each class is:

- Benign class: 961 not malicious executables. These files have been taken from DikeDataset [84].
- 4 Trojan classes containing the following families: *Ramnit* - 1500 samples, *Zbot* - 1200 samples, *IcedId* - 1000 samples and *Trickbot* - 800 samples.
- 4 Ransomware classes containing the following families: *Virlock* - 800 samples, *Stop* - 1000 samples, *Magniber* - 800 samples and *Wannacry* - 1500 samples.

Figure 4.7 illustrates each of these value for each family.

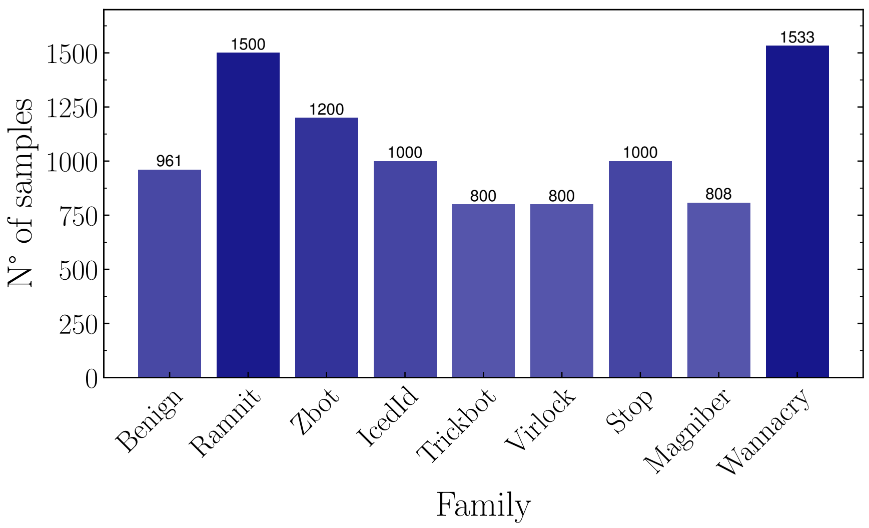


Figure 4.7. Number of samples per families contained in the ST-WinMal dataset.

### 4.3.2 ST-AndMal collection

Building an Android collection resulted to be more difficult than the Windows one, since VirusShare and MalwareBazaar seem to contain mostly Windows malware and they do not have enough Android samples for the selected families. Even the datasets mentioned in Section 4.2.1 are not well balanced in the number of samples per family and this could lead to a non-reliable classification if used in a ML system, hence we decided to keep this as a simple collection that will be later used to test the feature extraction by static and dynamic techniques. This collection contains:

- 4 Adware families: *Ewind* - 10 samples, *MobiDash* - 10 samples, *HiddenAd* - 288 samples, *Kuguo* - 200 samples.
- 4 Trojan families: *Hqwar* - 179 samples, *Anubis* - 46 samples, *Svpeng* - 23 samples, *Asacub* - 29 samples.

Unfortunately, even if 10 families of RiskTool type have been considered, there were not available samples on the sources used. Figure 4.8 illustrates the number of samples collected for each family.

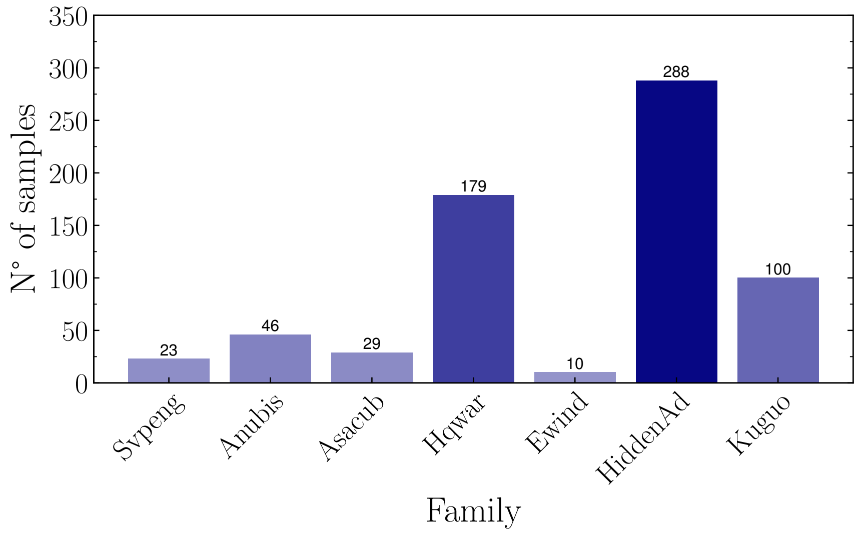


Figure 4.8. Number of samples per families contained in the ST-AndMal collection.

## 4.4 Comparison with other datasets

Despite many other datasets have already been created and made accessible in the past, such as the ones mentioned in Sec. 4.2.1, they were not perfectly suitable for our purposes, hence some of them had to be discarded or just partially used for the creation of our collections. This is due to the fact that most of the well known datasets, such as the one used for the Microsoft classification challenge [38], have been published to be used directly in a ML environment, hence containing only already extracted features or, as in the Microsoft's case, they contain samples that have been manipulated, removing their headers, to prevent execution. Our interest was to obtain samples that could have been ran and analyzed, leaving to us the task of extracting information. Other than that, we wanted to focus on families that are currently widespread and to collect samples that are in specific file formats (PE32 and APK), avoiding to take into consideration samples belonging to families that are not so common anymore or that comes in other formats and that should be treated in different ways.

# Chapter 5

# Analysis design

The next phase of this research, after having obtained samples of different families from the available sources, is to investigate them through *static* and *dynamic* analysis techniques in order to find common and different characteristics that could help in the characterization process. Rather than doing this through manual analysis, we preferred to take a more automatic approach, performing analysis on the collections described in the previous chapter with the goal of identifying meaningful information (i.e. features) and extract it. Once this information has been extracted it will be further analyzed to find similarities and differences and then prepared to become the input of a Machine Learning system to perform automatic characterization.
This chapter will present the analysis design, showing what has been used to investigate on the many behaviors each family could present, giving at first an initial presentation of the analysis system's workflow and structure. Then a more detailed description of the actual implementation of our system will be presented along with the features that have been selected for extraction.

## 5.1 Family study

Before starting an actual development of an analysis methodology, we decided to take some time to study malware behaviors, focusing on the families that were selected by consulting AV reports, in order to get an initial understanding of how malicious action could be performed and use that knowledge to model our system. During this process, we looked for detailed information on what a particular sample of a specific family does, by reading general family descriptions provided by AVs as well as reading technical reports of analyses performed on specific samples. We tried to collect as much information as possible on every family we had selected but, for some of them, there was not much information and it is also important to underline that during the writing of these thesis the statistics used to perform a selection change a little, hence some of the families that were studied in the first phase were not included in the dataset, leaving space to more "current ones".
The following sections will present a detailed description about the behavior of some samples which belong to the families contained in the dataset, trying to emphasize those actions that are used to perform and achieve **persistence**, communication with a **Command&Control server** (C&C) and **obfuscation**.

### 5.1.1 Zbot

Zbot can be installed by other malware families such as Cutwail, Dofoil etc, but usually it is received by spam emails. Once running on a system, it looks for the Remote Desktop Service (RDS) and tries to run a process for every RDS session, creating a copy of itself in the startup folder. There are some variants which also drop copies on the system folder as well as some encrypted files containing stolen data. Zbot tries to perform code injection into every running process which matches the privilege level of the current logged user. It also performs API hooking of some Windows APIs to intercept sensitive data. In order to perform malicious actions undetected, it also tries to lower Internet Explorer and Firefox security (i.e. disabling warnings for insecure pages). It communicates with C&C servers that can send configuration files used to customise its behavior. For example, these configuration files can contain [86]:

- Locations from which to download updates for the malware.
- Online financial institution that shall be targeted.
- JavaScript payload that can be used to steal data.
- Targeted browsers.

### 5.1.2 Trickbot

Trickbot samples are usually small in size (less than 500KB) and it does not use any additional packaging or encryption for the main body. It receives a list of commands from a C&C server, which are contained in an AES-encrypted file. When it is launched for the first time it creates an array of encrypted functions on the stack that will be decrypted at run-time, executed and encrypted again. It tries to perform UAC bypass to obtain admin privileges on the system. One particular technique that is used to achieve stealthiness is the Heaven's gate [87] which permits to switch from 32-bit to 64-bit code without issuing a proper system call that would let the OS handle this transition. Trickbot uses scheduled tasks to achieve persistence, which is usually called *AudibleFree*. It establishes TLS connections with C&C servers to which it sends collected information [88, 89]

### 5.1.3 Ramnit

This trojan tries to perform code injection into some processes by loading a device driver. Some example of targeted processes are *winlogon.exe* and *iexplore.exe*. It also connects to remote servers in order to be instructed on commands and actions that should be executed [90]. It is capable of performing many malicious actions such as [91]:

- MITM attacks.
- Domain Generation Algorithm (DGA) used to find the C&C server.
- Privilege escalation.
- Lower AntiVirus security by adding exceptions.
- Uploading screenshots containing sensitive information.

The main binary is packed using UPX compression and custom packing. Once unpacked it mainly consists of three general functions:

1. **ApplyExploit**: it is used to perform privilege escalation exploiting some known vulnerability such as CVE-2013-3660 and CVE-2014-4113; after checking it out the system is actually affected by those vulnerabilities.
2. **CheckBypassed**: once checked that, the binary is running with admin privileges, adding registry keys to avoid detection (i.e. obfuscation/evasion).
3. **start**: it is a routine that coordinates the two previous functions and, once they have been executed succesfully, it creates two *svchost.exe* processes and writes two dlls.

### 5.1.4 Anubis

Anubis is an Android banking trojan which is capable of stealing information performing Overlay attacks and intercepting Calls/SMSs. It can also act as a keylogger. It is generally delivered through third-party sites but some variant have also been found on the official Play Store. It is composed by two modules: a downloader and a payload. The latter is obtained from a C&C server. This malware family achieves persistence using the permission *RECEIVE_BOOT_COMPLETED* which allows to open the app each time the mobile device is restarted. It also avoid deletion by showing error messages when the user tries to uninstall it by intercepting clicks and monitoring events. It sends device information to the C2 server: this information includes the list of installed applications, which is used by the server to send customized payloads and pages that will be used to perform overlay attack and steal sensitive data. It also intercepts SMSs, which allows it to steal OTP codes. It requests to become the default SMS app on the device which will allow to also remove received messages and hide malicious actions from the user. Other than SMS, phone calls can be intercepted and redirected as well [92].

### 5.1.5 Svpeng

This trojan exploits accessibility services, which are meant to provide UI enhancements for user that cannot interact fully with a device. By abusing this feature, the trojan will be able both to steal user data and gin higher privileges, obtaining administrative permissions that will permit it to prevent being deleted. It takes screenshots of pages containing sensitive data or it performs an overlay attack if this action is denied by the target application. It receives commands from C&C server. It is generally distributed by malicious third party sites as a fake flash player [93].

### 5.1.6 Asacub

Asacub has been detected for the first time in 2015. Back then it was a spyware family, known as Smaps, and it evolved later into a banking trojan [94]. It communicates with C2 servers, the communication is encrypted with RC4, meanwhile its ancestor spyware version used to send everything in clear. This malware family propagates by sending SMSs containing a link to a web-page where the malware can be downloaded. It stresses the user by prompting access request pages for Accessibility services until these are given and then it will set itself as the default app and disappears from the main screen. C&C server sends commands which may differ between the many flavors of this family. It heavily employs obfuscation techniques, such as string concatenation, classes and methods renaming to implement functions in native code (C/C++) which would require the analyst to use different tools able to deal with those files [95].

### 5.1.7 Hqwar

Firstly appeared in 2016 [96], Hqwar is a banking trojan which mainly acts as a dropper for other families. It contains a *.dex* file encrypted with RC4. Actually, it does not drop any application, but it loads it, thus avoiding to ask for the victim's permissions.

### 5.1.8 Stop

This Windows ransomware family usually spreads through pdf files attached in spam e-mails. It implements two persistence mechanisms: an *AutoRun* registry key and a scheduled task created using COM objects. It also downloads other malicious files. Encryption targets both local drives and network shares, and it uses the Salsa20 algorithm. It droppes copies of itself in other locations, tries to connect to the internet to obtain localization data. Once the encryption process has been completed, it drops a ransom note with instructions on how to pay the ransom [97].

### 5.1.9 Wannacry

This famous ransomware family spreads throgh a worm component, exploiting a well-known vulnerability called EternalBlue [16]. Once running on the victim's system, it starts a three-stage attack:

1. Contacts a kill witch domain, if it does not receive a response it will drop its worm component which will replace the executable *tasksche.exe*. If the kill switch responds, the malware will stop its actions.

2. Drops some files (dlls and executables) that are needed to perform encryption.

3. Creates multiple threads dedicated to encrypt victim's files.

It obtain persistence by running a service named *mssecsvc.exe* and adding a registry key named *WannaCrypt0r*. In order to avoid the restoring of the system to a previous state, it deletes backups and tries to prevent the system to being booted in safe mode [98].

### 5.1.10 Comparison between families characteristics

Tables 5.1.10, 5.1.10 and 5.1.10 summarize the descriptions provided above in the previous sections.

| | **Zbot** | **Trickbot** | **Ramnit** |
|---|---|---|---|
| **Persistence** | Copies itself in other locations | Scheduled task | Registry keys, Autorun task |
| **Techniques** | API hooking | UAC bypass, Heaven's gate | MITM, DGA, privilege escalation |
| **Spreading** | Spam emails, installed by other families | ? | Email attachments |
| **C&C server** | Receives configuration files | Receives commands | Receive commands |

Table 5.1. Common and different characteristics for Windows Trojan families.

| | **Stop** | **Wannacry** |
|---|---|---|
| **Persistence** | Registry keys, background task | Registry keys, deletes shadow copies |
| **Techniques** | Encrypts with Salsa20 | Encrypts with AES, EternalBlue exploit |
| **Spreading** | Email attachments | Worm component |
| **C&C server** | ? | Kill switch |

Table 5.2. Common and different characteristics for Windows Ransomware families.

| | **Anubis** | **Svpeng** | **Asacub** | **Hqwar** |
|---|---|---|---|---|
| **Persistence** | Permission to start on boot | Prevents uninstalling | ? | ? |
| **Techniques** | Keylogging, overlay attack | Steals SMSs and calls, screenshots | Installs as SMS app | Dynamically loads its modules |
| **Spreading** | Third party sites, Play store | Fake flash player on malicious sites | Phishing SMSs | ? |
| **C&C server** | Receives payloads | Receives commands | Receives commands | ? |

Table 5.3. Common and different characteristics for Android Trojan families.

## 5.2 System architecture

In order to fulfill our purposes, we designed an analysis system which leverages on well-known tools to extract meaningful information from each sample of our collections. Fig. 5.1 depicts its structure. The system is divided into an Android and a Windows part which takes samples from the relative datasets and extract some features. The following sections will dive into the workflow and the implementation of each component, showing which tools have been used and which are the features that this system aims to extract.

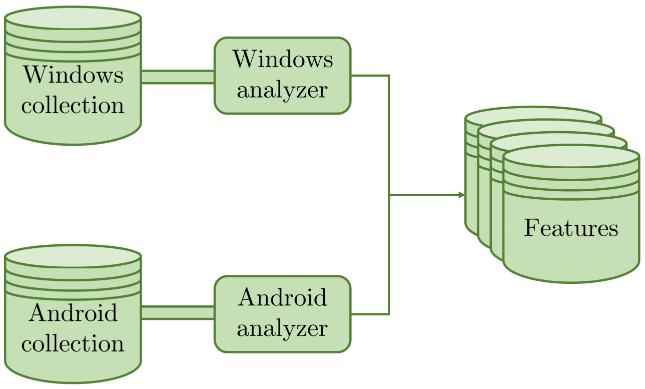


Figure 5.1. Overall architecture.

### 5.2.1 Windows analyzer

The Windows analyzer is in charge of applying analysis techniques to obtain useful information from samples. Even if dynamic techniques could produce more meaningful and representative information, since the malware is actually executed and its behavior is monitored, it requires a consistent amount of computational resources in order to be applied to a large number of samples. Running dynamic analysis on a large collection of samples basically requires two prerequisites:

1. Enough computational resources: the analyst should setup an isolated and monitored system, usually known as a *sandbox*, where the malware can be ran and its behavior tracked.

2. Enough time: generally speaking, the more a sample can be monitored while executing, more information will be generated. In order to test thousand of samples in a feasible amount of time the best option is to have more sandboxes running concurrently which, once again, requires resources.

For these reasons the analyzer combines both *static* and *dynamic* approaches on samples, applying the latter only to a subset of the dataset and the first to the whole collection.

### Static analyzer

This component of the Windows analyzer is in charge of parsing a sample and, without executing it, extract the information that is contained inside. It mainly relies upon the *pefile* Python library which extracts information from PE headers and sections. Fig. 5.2 depicts its structure. It aims to extract three types of information from each sample:

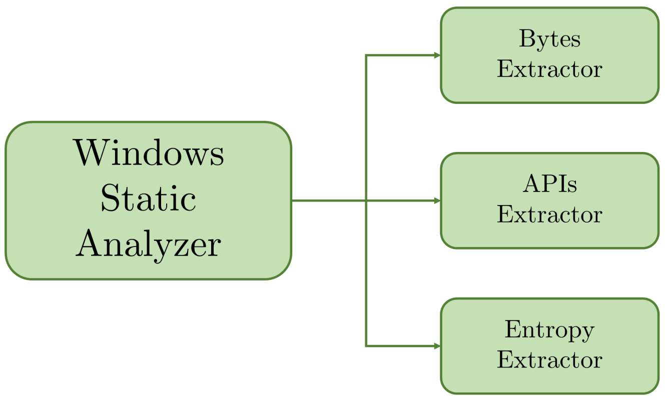


Figure 5.2. Windows static analyzer structure.

- **Bytes**: the *Bytes Extractor* is in charge of removing PE headers and just dumping all the section bytes in a *.bytes* file. This files can be used to perform n-gram analysis and it also makes the sample sterile, since it cannot run anymore, preserving all the information contained in its sections.
- **APIs**: the *APIs Extractor* extracts all the imported APIs. These are retrieved by looking at the Import Address Table (IAT), if available.
- **Entropy**: the *Entropy extractor* takes all the PE's sections and calculate their entropy. This is done both for standard sections such as *.text .data* etc. and for non-standard sections (i.e. sections that may have been defined by the malware author himself, these may include additional code and data).

### Dynamic analyzer

As already mentioned before, dynamic analysis requires to setup an isolated and monitored environment where a malware sample can run freely. We sat up such an environment using the *Cuckoo Sandbox* [69]. This sandbox basically consist of an Host where the main Cuckoo components run, and an Agent which runs inside a virtual machine (VM). The agent environment was prepared by creating a Windows 7 VM running on VirtualBox. This machine was configured to use the

*Host-only* network mode, which allows only communication between guest and host. In addition to that, we added **InetSim**, a network services simulator, which is used to further deceive samples into thinking that they are running with a working connection. Fig. 5.3 illustrates the environment structure.

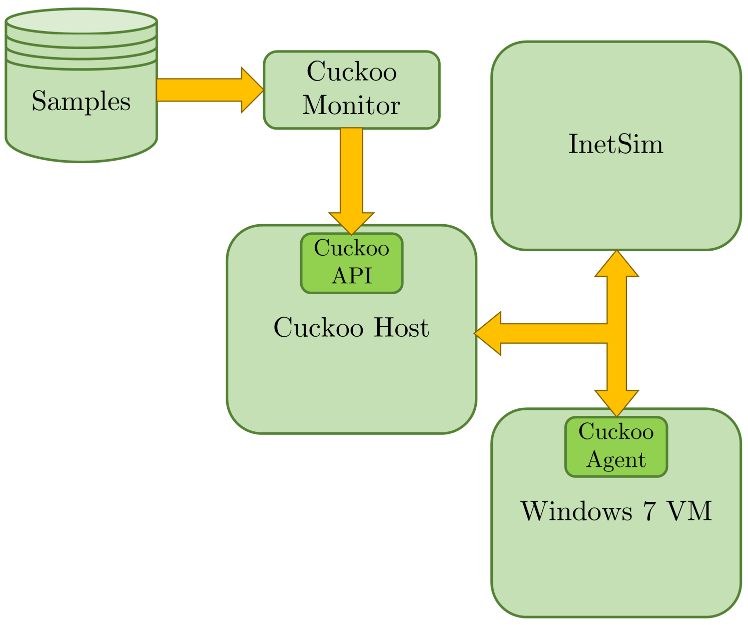


Figure 5.3. Windows dynamic analyzer structure.

The **Windows 7 VM** runs the Cuckoo agent, which practically is a Python program that once launched listens for commands from the host. The host will send the sample that will be injected and ran inside the VM, meanwhile many Cuckoo components will be monitoring its actions. Once the analysis is completed (i.e. the time specified is expired or some error has occurred) all this information will be saved on the host and the VM will be restored to a previous snapshot where the sample was not yet injected. In order to automate such an activity we developed the **Cuckoo Monitor** which is a Python program that stands between our collection of samples and the Cuckoo host instance, interacting with the latter using Cuckoo API. This is another feature offered by Cuckoo which makes the host listening for commands on specific endpoints, giving the possibility to distribute the whole environment since the API can be placed on an endpoint reachable from the outside. The Monitor does the following actions:

1. **Submission phase**: for each family in our collection, a prefixed number of samples is taken and submitted to the Host, for each successful submission a *taskId* is returned that will be saved. Then the monitor sleeps for two minutes.

2. **Waiting phase**: once awaken the monitor will ask for the reports relative to the submitted tasks, going back to sleep for 10 seconds if none is available.

3. **Parsing phase**: when an analysis is completed its report is retrieved (which is a JSON document). From this report the API trace is extracted; this contains the API used by the sample during execution along with the number of usages.

4. **Cleaning phase**: every file generated by Cuckoo is deleted. This is done to prevent memory saturation.

This process is repeated until a desired amount of result is achieved. For each sample that has produced valid result (i.e. the dynamic report has useful information and it is not empty) a counter is updated. It may happen that some samples don't produce any results (i.e. the malware has detected it is being analyzed or a generic Cuckoo error happened), in this case the counter will not be increased, hence more samples will be submitted until the desired amount of result has been reached or there are no more samples available. Figure 5.4 depicts the monitor components used in the four phases described above.

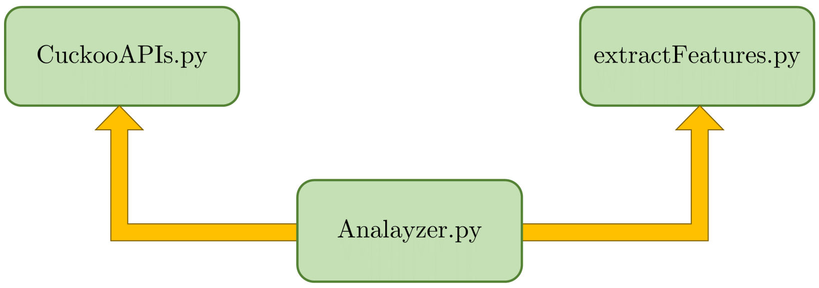


Figure 5.4. Cuckoo Monitor's structure.

The monitor basically consist of three Python modules:

- *cuckooAPIs.py*: it is the point of access to the Cuckoo's API. It implements the methods needed to perform some basic actions such as submitting a sample, retrieving and deleting a report and so on.
- *extractFeatures.py*: this module implements the extraction of APIs from the report. Actually this file also contains method which belongs to the static analyzer to extract static features, but these methods can work with Cuckoo as well if the analyst wants to retrieve also static information from the generated report.
- *analyzer.py*: this is the core component of the monitor which implements the four phases described before by interacting with the other two modules.

### 5.2.2 Android analyzer

The Android analyzer presents a similar structure to the Windows one, being composed of a static and a dynamic part that are meant to extract different count of features. Since the number of samples contained in the Android collection is smaller compared to the Windows' one, here both static and dynamic analysis are performed on the whole collection, differently from the Windows analyzer, where the dynamic part was used only on a subset of the dataset.

#### Static analyzer

The Android static analyzer relies upon *Droidlysis* [99] to extract information from APKs. It tries to extract three types of features from each sample. Fig. 5.5 illustrates its structure.

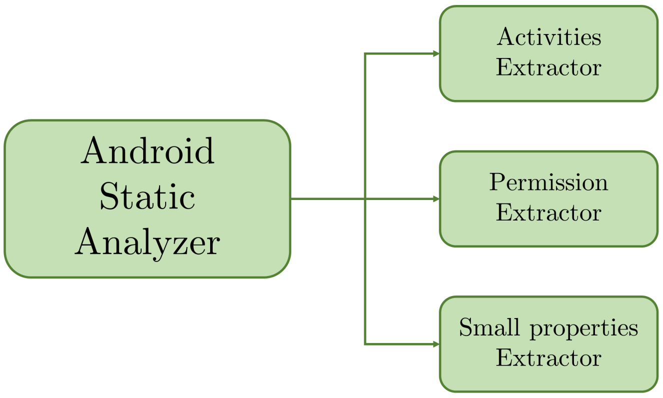


Figure 5.5. Android static analyzer structure.

- **Activities**: they are basically the windows where the application draw its UI and they are one of the main components of an application.
- **Permissions**: they basically tell what the application intends to do, asking access to particular features and/or resources of the system.
- **Smali properties**: these are inferred from droidlysis, they are a description of what the Dalvik code (the one contained in the .dex files) tries to do, such as sending sms, intercepting phone calls etc.).

### Dynamic analyzer

In order to perform dynamic analysis, we used CuckooDroid, which is an extension of Cuckoo that can be used to analyze Android applications. Once again the Cuckoo agent will be placed inside a VM, a Linux one this time, where an Android emulator will be in charge of emulating an Android device where applications will be ran. Fig. 5.6 depicts the analyzer architecture.

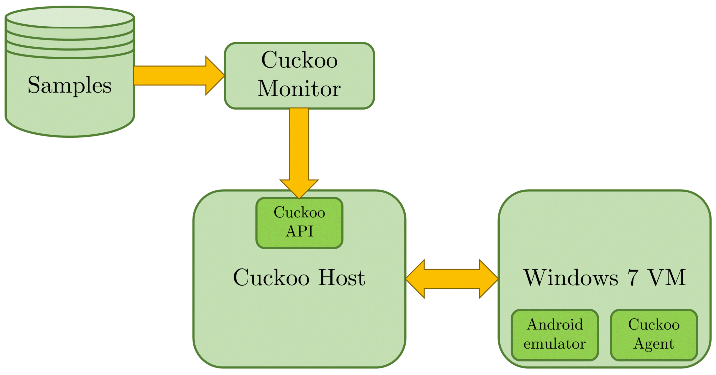


Figure 5.6. Android dynamic analyzer structure.

Since CuckooDroid is just an extension of Cuckoo, the basic functionalities used for Windows are still available, hence it was not necessary to write a complete new system but only some functions were added (the ones to handle the reports, which are obviously different). The workflow of the Cuckoo's monitor is the same of the Windows' one (it goes cthrough the same four phases) and also the Monitor structure remains the one depicted in Fig. 5.4.

## 5.3 Tools

Each component of the system described above relies on some well-known analysis tool to achieve its goal. The following section will give more specific details on what are the tools that have been used by each component to fulfill the task of feature extraction and how these have been actually used, providing some usage examples as well.

### 5.3.1 Windows static analyzer

The static analyzer for Windows samples, which are only executables (no dll, documents etc.), aims to extract three kind of features: *entropy* of each section, *bytes* of the executable (headers excluded) and *APIs* imported. All these kinds of information are embedded inside the executable and are extracted using *pefile* [100], which is a Python module able to parse Windows executable and inspect the information they contain.

#### Extracting bytes

This feature is the easiest to extract, since it can be done by just reading the file bytes. This would have left each sample as it is so to guarantee sterility, headers are removed. This is done by obtaining the address of the first section of the executable. Listing 5.1 reports the code used to obtain this address.

```python
import pefile
def readBytes(source):
    f = open("source", "r")
    pe = pefile.PE(source, fast_load=True)
    #get address of the first section
    addr = pe.sections[0].PointerToRawData
    if addr == 0:
        print("Address not found")
        return -1
    pe.close()
    f.seek(addr)
    readBytes(f)
```

Listing 5.1. Code used to obtain a pointer to the first section.

#### Extracting entropy

Each executable may come with different sections, in fact, apart from the standard ones, malware authors can insert custom sections. This can be done to hide some portion of the code or of data that will be later used. Using pefile it is possible to get a list of all section and obtain their entropy. Listing 5.2 shows an example of code which extracts the entropy of all the sections contained in a PE.

```python
import pefile
def extractSectionEntropy(source):
    #dictionary where the pair <key,value> is <section, entropy>
    e_d = {}
    pe = pefile.PE(source, fast_load=True)
    for entry in pe.sections:
        e_d[str(entry.Name.strip(b"\x00"))] = entry.get_entropy()
    return e_d
```

Listing 5.2. Code used to obtain entropy of each section.

### Extracting APIs

Except for some particular case, each executable carries with it a list of imported functions that will be used during execution. This functions can be found inside the Import Access Table (IAT), which can be accessed with pefile by the attribute *DIRECTORY_ENTRY_IMPORT*. Once extracted, the analyzer saves them in a file named "hash.apis", where "hash" is the sha256 or md5 of the sample. Listing 5.3 shows an usage example where the IAT is accessed to obtain imported APIs.

```python
import pefile

def extractStaticAPIs(source):
    imports = []
    pe = pefile.PE(source)
    for entry in pe.DIRECTORY_ENTRY_IMPORT:
        for imp in entry.imports:
            if imp != None:
                imports.append(imp.name.decode())
    return imports
```

Listing 5.3. Code used to extract APIs from the IAT.

## 5.3.2 Windows dynamic analyzer

Dynamic analysis is performed through the Cuckoo sandbox [69], with the goal of obtaining the APIs used by the sample during execution as well as the number of times each of them is called. As already mentioned previously, Cuckoo works by running its main component on an host system and an agent which runs inside a VM where the sample will run. These two components talk to each other using the XMLRPC protocol. In our analysis system Cuckoo's host is placed on an Ubuntu 20.02 operating system, meanwhile the agent runs inside a Windows 7 VM. Tab. 5.3.2 reports VM's configuration. The Windows VM has been created using VirtualBox.

| **Windows 7 VM** | |
|---|---|
| Name | cuckoo1-win7 |
| Guest OS | Windows 7 (64 bit) |
| Memory size | 4096 MB |
| Number of CPUs | 2 |
| Shared folders | None |
| Network type | Host-only |

Table 5.4. Configuration of the Windows 7 Virtual Machine.

Cuckoo can use different tools to perform analysis. Once installed it is possible to customise the analysis through some configuration files. In order to get behavioral analysis, which will capture any API used, the *processing.conf* file needs to be modified, enabling the *behavior* section. In some initial attempts we also enabled the memory dump, which requires the installation of Volatility, but this generated huge reports (3-4 GB each) therefore we decided to turn this option off. After filling the configuration files there are three component of Cuckoo that needs to be launched:

- **Cuckoo**: it is the central component, it will look for declared machines and then it waits until an analysis is submitted.
- **Rooter**: it is required to perform some additional commands on Cuckoo such as using InetSim to simulate working network services.
- **Api**: it creates some endpoints that can be contacted to send commands to the central component. These API only accepts packets which contain a token in the header, which can be found in the configuration files.

Once these components are running it is possible to perform analysis. This can be done by submitting a sample (by the api or the *cuckoo submit* command). When a sample is submitted Cuckoo returns a *taskId* which is a number identifying the analysis. This id can be later used to retrieve the report or get the task state. Each task goes through four states:

1. **Submitted**: the sample is put in a queue until Cuckoo is ready to inject it inside the VM.
2. **Running**: the sample is running inside the monitored VM.
3. **Completed**: analysis is finished and the report files are being generated.
4. **Reported**: the report is finally available.

When the analysis is complete (i.e. task is in the reported state), Cuckoo generates a result folder in the *.cuckoo/storage/analyses* directory. The only interesting file for our purposes is the *report.json* which contains a summary of the analysis and a description of various actions the sample has performed. Listing 5.4 shows a section of a report obtained on a Wannacry sample, which reports commands executed through command line.

```
{
        "families": [],
        "description": "Command line console output was observed",
        "severity": 1,
        "ttp": {},
        "markcount": 3,
        "references": [],
        "marks": [
        {
            "families": [],
            "description": "Tries to locate where the browsers are installed",
            "severity": 1,
            "ttp": {},
            "markcount": 1,
            "references": [],
            "marks": [
                {
                    "category": "file",
                    "ioc": "c:\\program files\\mozilla firefox\\firefox.exe",
                    "type": "ioc",
                    "description": null
                }
            ],
            "name": "locates_browser"
        },
        {
            "families": [],
            "description": "Checks amount of memory in system, this can be used
                 to detect virtual machines that have a low amount of memory
                available",
            "severity": 1,
            "ttp": {
                "T1082": {
                    "short": "System Information Discovery",
                    "long": "An adversary may attempt to get detailed
                        information about the operating system and hardware,
                        including version, patches, hotfixes, service packs, and
                         architecture."
                }
            }
```

Listing 5.4. Section of a Cuckoo report.

### 5.3.3 Android static analyzer

The static analyzer uses Droidlysis [99], which is a property extractor for Android applications, that is able to extract properties from APK as well as disassembling application's code and retrieve some information about what it does. Droidlysis leverages on other tools to extract more information, these can be specified by editing a configuration file, named *droidconfig.py*. We configured it to use:

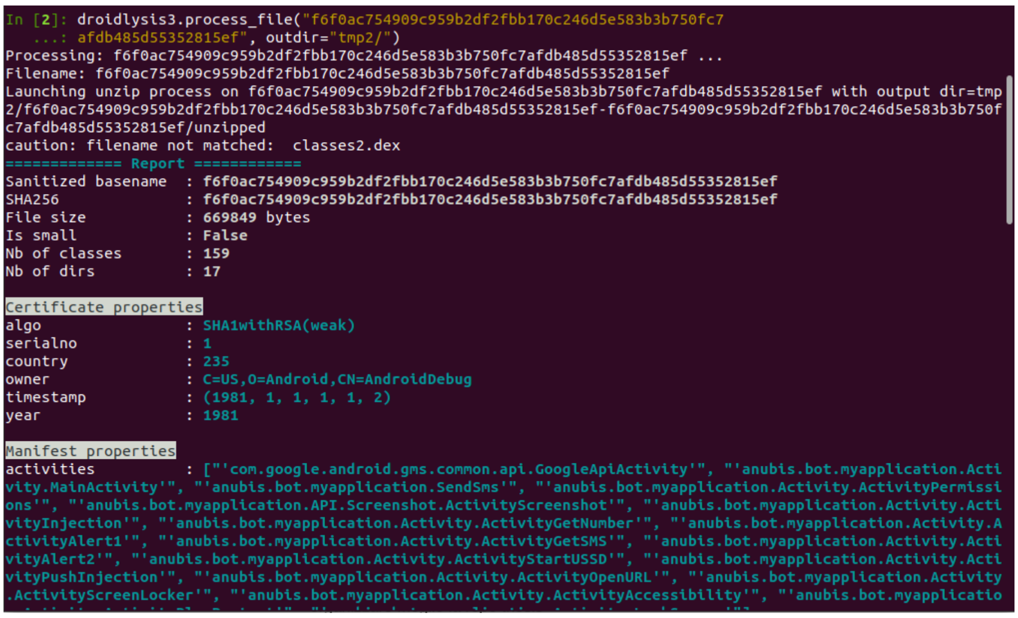
- **Apktool**: it can unpack and repack android packages, it is used to extract resources from samples.
- **Baksmali**: it is a disassembler for the dex files, which are the ones containing the application's code.
- **Dex2Jar**: can convert dex files to class files, it also has a disassembler but with a different implementation from baksmali.

When droidlysis is launched it prints out the extracted information as well as producing a more detailed report in a specific folder, which can be specified when launching the analysis. Analyses can be start in two ways: by executing the central Python script (*droidlysis.py*) or by importing droidlysis as a library in a Python program. Listing 5.5 shows how to launch an analysis, meanwhile Figs. 5.7 and 5.8 show an example of the output generated by Droidlysis on a sample which belongs to the Anubis family. From these it is possible to see that this tools provides some basic information such as file's hash, its size, class number and so on, as well as more useful information, such as the features we are looking for.

```python
import droidlysis3

droidlysis3.process_file(sample, outdir="tmp/")
```

Listing 5.5. Code used to launch droidlysis from Python.

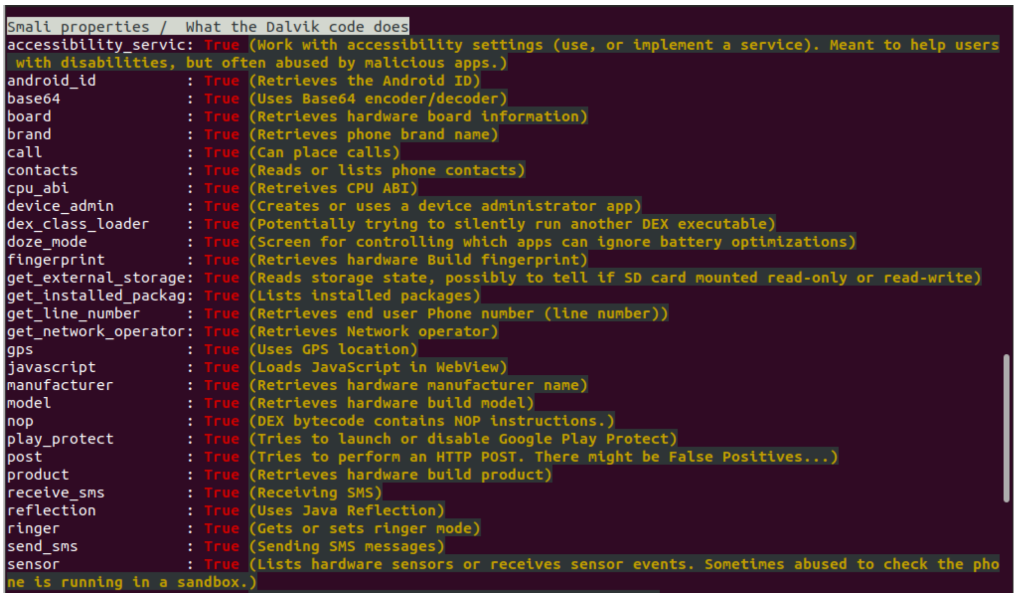


```
In [2]: droidlysis3.process_file("f6f0ac754909c959b2df2fbb170c246d5e583b3b750fc7
   ...: afdb485d55352815ef", outdir="tmp2/")
Processing: f6f0ac754909c959b2df2fbb170c246d5e583b3b750fc7afdb485d55352815ef ...
Filename: f6f0ac754909c959b2df2fbb170c246d5e583b3b750fc7afdb485d55352815ef
Launching unzip process on f6f0ac754909c959b2df2fbb170c246d5e583b3b750fc7afdb485d55352815ef with output dir=tmp
2/f6f0ac754909c959b2df2fbb170c246d5e583b3b750fc7afdb485d55352815ef-f6f0ac754909c959b2df2fbb170c246d5e583b3b750f
c7afdb485d55352815ef/unzipped
caution: filename not matched:  classes2.dex
============= Report ============
Sanitized basename   : f6f0ac754909c959b2df2fbb170c246d5e583b3b750fc7afdb485d55352815ef
SHA256               : f6f0ac754909c959b2df2fbb170c246d5e583b3b750fc7afdb485d55352815ef
File size            : 669849 bytes
Is small             : False
Nb of classes        : 159
Nb of dirs           : 17

Certificate properties
algo                 : SHA1withRSA(weak)
serialno             : 1
country              : 235
owner                : C=US,O=Android,CN=AndroidDebug
timestamp            : (1981, 1, 1, 1, 1, 2)
year                 : 1981

Manifest properties
activities           : ["'com.google.android.gms.common.api.GoogleApiActivity'", "'anubis.bot.myapplication.Acti
vity.MainActivity'", "'anubis.bot.myapplication.SendSms'", "'anubis.bot.myapplication.Activity.ActivityPermissi
ons'", "'anubis.bot.myapplication.API.Screenshot.ActivityScreenshot'", "'anubis.bot.myapplication.Activity.Acti
vityInjection'", "'anubis.bot.myapplication.Activity.ActivityGetNumber'", "'anubis.bot.myapplication.Activity.A
ctivityAlert1'", "'anubis.bot.myapplication.Activity.ActivityGetSMS'", "'anubis.bot.myapplication.Activity.Acti
vityAlert2'", "'anubis.bot.myapplication.Activity.ActivityStartUSSD'", "'anubis.bot.myapplication.Activity.Acti
vityPushInjection'", "'anubis.bot.myapplication.Activity.ActivityOpenURL'", "'anubis.bot.myapplication.Activity
.ActivityScreenLocker'", "'anubis.bot.myapplication.Activity.ActivityAccessibility'", "'anubis.bot.myapplicatio
```

Figure 5.7. Output of Droidlysis.



```
Smali properties /  What the Dalvik code does
accessibility_servic: True (Work with accessibility settings (use, or implement a service). Meant to help users
 with disabilities, but often abused by malicious apps.)
android_id          : True (Retrieves the Android ID)
base64              : True (Uses Base64 encoder/decoder)
board               : True (Retrieves hardware board information)
brand               : True (Retrieves phone brand name)
call                : True (Can place calls)
contacts            : True (Reads or lists phone contacts)
cpu_abi             : True (Retreives CPU ABI)
device_admin        : True (Creates or uses a device administrator app)
dex_class_loader    : True (Potentially trying to silently run another DEX executable)
doze_mode           : True (Screen for controlling which apps can ignore battery optimizations)
fingerprint         : True (Retrieves hardware Build fingerprint)
get_external_storage: True (Reads storage state, possibly to tell if SD card mounted read-only or read-write)
get_installed_packag: True (Lists installed packages)
get_line_number     : True (Retrieves end user Phone number (line number))
get_network_operator: True (Retrieves Network operator)
gps                 : True (Uses GPS location)
javascript          : True (Loads JavaScript in WebView)
manufacturer        : True (Retrieves hardware manufacturer name)
model               : True (Retrieves hardware build model)
nop                 : True (DEX bytecode contains NOP instructions.)
play_protect        : True (Tries to launch or disable Google Play Protect)
post                : True (Tries to perform an HTTP POST. There might be False Positives...)
product             : True (Retrieves hardware build product)
receive_sms         : True (Receiving SMS)
reflection          : True (Uses Java Reflection)
ringer              : True (Gets or sets ringer mode)
send_sms            : True (Sending SMS messages)
sensor              : True (Lists hardware sensors or receives sensor events. Sometimes abused to check the pho
ne is running in a sandbox.)
```

Figure 5.8. Output of Droidlysis.

### 5.3.4 Android dynamic analyzer

Dynamic analysis on Android samples is performed through CuckooDroid, an extension of the Cuckoo sandbox which allows to submit android applications and analyze them. Once again, CuckooDroid uses one central component and an agent which communicates to each other through the XMLRPC protocol. In this case it is needed to emulate and Android device in order to run applications. CuckooDroid offers three implementation choices:

- **Android on Linux machine**: both the agent and the emulator run on a Linux VM. After an analysis the system is restored to its previous state by the means of snapshots, in the same way as it is done for the standard Cuckoo Sandbox.
- **Android emulator**: host, agent and emulator run simultaneously on the same system. Each time an analysis is performed the emulated device is duplicated to have a clean state (i.e. snapshots only on the emulator, not the entire system).
- **Android device cross-platform**: here the android device is virtualized (there is not a VM which runs an emulator) and the agent is not a python program anymore, but it becomes an APK which runs inside the virtualized device.

Among these three choices we choose the first one since it is more similar to the Windows' implementation and it was easier to configure. Tab. 5.3.4 reports the details of the Linux VM.

| **Linux VM** | |
|---|---|
| Name | cuckoodroid-14-aol |
| Guest OS | Ubuntu 14.04 (64 bit) |
| Memory size | 16384 MB |
| Number of CPUs | 8 |
| Shared folders | None |
| Network type | Host-only |

Table 5.5. Configuration of the Linux Virtual Machine.

Once the VM and the emulator are running there are some applications that need to be installed, these will prevent malware samples from knowing that they are running inside an emulator as well as capturing actions that the sample will perform while it is running. These applications are provided with CuckooDroid and the documentation explains how to correctly install them. Samples can be submitted more or less in the same way as per Cuckoo, using the submit command or by running the API service and contacting the proper endpoint. Differently from the Windows analyzer, here the Cuckoo rooter is not needed (it was necessary in the Windows case since we were using InetSim). Everything else remains unchanged: each submission generates a taskId, each task will go through four states and once the analysis is completed a json and an html report are generated. These reports have both static and dynamic information as can be seen in Listings 5.6, 5.7 that show two sections of a report generated from an Anubis sample.

```
{
        "families": [],
        "description": "Application Asks For Dangerous Permissions (Static)",
        "severity": 3,
        "references": [],
        "alert": false,
        "data": [
            {
                "process": null,
                "signs": [
                    {
                        "type": "android.permission.ACCESS_FINE_LOCATION",
                        "value": "Access fine location sources, such as the
                            Global Positioning System on the phone, where
                            available. Malicious applications can use this to
                            determine where you are and may consume additional
                            battery power."
                    }
                ]
            }
```

Listing 5.6. Example of a CuckooDroid report's section which shows some static information.

```
"droidmon": {
    "api": {
        "android_telephony_TelephonyManager_getLine1Number": 4,
        "android_util_Base64_encode": 12,
        "android_app_SharedPreferencesImpl_EditorImpl_putString": 218,
        "android_app_ActivityThread_handleReceiver": 29,
        "android_content_ContextWrapper_startActivity": 396,
        "android_util_Base64_encodeToString": 6,
        "java_net_URL_openConnection": 9,
        "android_telephony_TelephonyManager_getNetworkOperatorName": 4,
        "android_util_Base64_decode": 444,
        "java_io_File_exists": 306,
        "android_app_ApplicationPackageManager_setComponentEnabledSetting": 1,
        "java_net_ProxySelectorImpl_select": 10,
        "android_telephony_TelephonyManager_getNetworkCountryIso": 4,
        "libcore_io_IoBridge_open": 78,
        "android_app_Activity_startActivity": 2,
        "android_content_ContextWrapper_startService": 145,
        "java_lang_reflect_Method_invoke": 1,
        "android_os_SystemProperties_get": 24
    }
```

Listing 5.7. Example of a CuckooDroid report's section which shows some dynamic properties.

## 5.4 Malware features

Previous researches [101, 102] have shown how malware may contain different kinds of information that can be used to identify and characterize them. Despite we were able to extract most of them for most of the samples, some of them have been discarded because they should be later used as input for a ML system, hence they should be suitable for this purpose. Some features, as strings, IP addresses contacted or the process tree, would require an appropriate encoding in order to become a valid input, which in some case could be quite complex and there is no guarantee that it would lead to better results. Having considered this, we performed a selection of features that our analysis system should extract, taking into consideration the ones that better suit our purposes. These are reported in Tabs. 5.6 and 5.7.

| **Windows features** | | |
|---|---|---|
| **Feature** | **Extracted through** | **Description** |
| Bytes | Static analysis | The actual bytes that composes the executable, can be used to perform n-gram analysis. |
| APIs | Static analysis | List of imported APIs. They are declared in the IAT. |
| Section entropy | Static analysis | Entropy of each section that composes the executable. |
| API count | Dynamic analysis | List of APIs actually used during execution, along with the number of usages. |

Table 5.6. List of selected features to be extracted from Windows samples.

| **Android features** | | |
|---|---|---|
| **Feature** | **Extracted through** | **Description** |
| Activities | Static analysis | The list of Activities that constitute the application. |
| Permission list | Static analysis | The list of permissions required by the app to run, they are declared in the manifest. |
| Smali properties | Static analysis | Properties of the smali code (i.e. what it tries to do). |
| API count | Dynamic analysis | APIs that are actually used, along with the number of usages. |

Table 5.7. List of selected features to be extracted from Android samples.

## 5.5 Comparison with other analysis tools

There are many researches focused on malware identification and characterization, which describe the development of systems comprehensive of both an analyzer and an ML classifier [5, 6, 7], where they perform identification or characterization, but using only a single type of features. R. B. Hadiprakoso et al. [103] have developed an Android malware detection system that uses combination of features, but they focused only upon identification (i.e. labelling a sample as benign or malicious). In comparison to these systems, we tried to build something that would give more flexibility to the analyst, allowing him to choose between different types of features, deciding whether to apply static and/or dynamic analysis. It is also important to outline that this research is not focusing at all on the creation of a ML classifier, but rather on the development of an analysis methodology that could aid the process of characterization, by understanding which kind of information can represent the behavior of a family, and extracting it.

# Chapter 6

# Results

## 6.1 Features Datasets

Starting from the Windows malware dataset described in Chapter 4 4.3.1, which contains only benign and malicious executables, we built other datasets that do not contain samples, but their features. These features can be used as a source of data to train a ML system, using each one of them one by one or combining them together. The whole dataset has been subjected to static analysis, while only a smaller subset has been processed through dynamic approaches.

### 6.1.1 Static features dataset

Each sample which belongs to our dataset has been processed with the Windows Static analyzer extracting three kind of features: *section bytes*, *imported APIs* and *section entropy*. Figs. 6.1, 6.2 and 6.3 show the number of features available for each class. This number may differ from the original number of samples processed since some samples had to be discarded when they did not contain valid results (i.e. an empty Import Access Table).

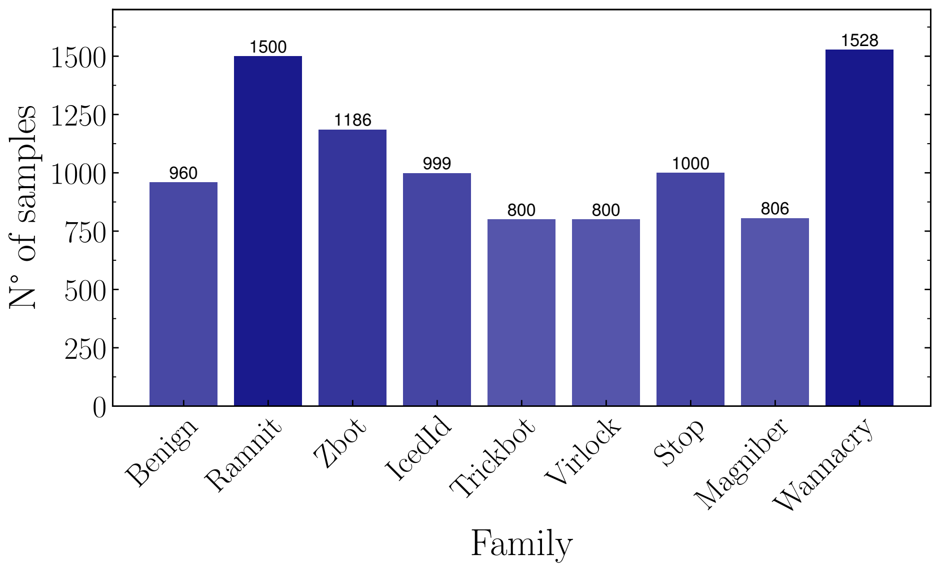


Figure 6.1. Number of samples per family for the *bytes* dataset.

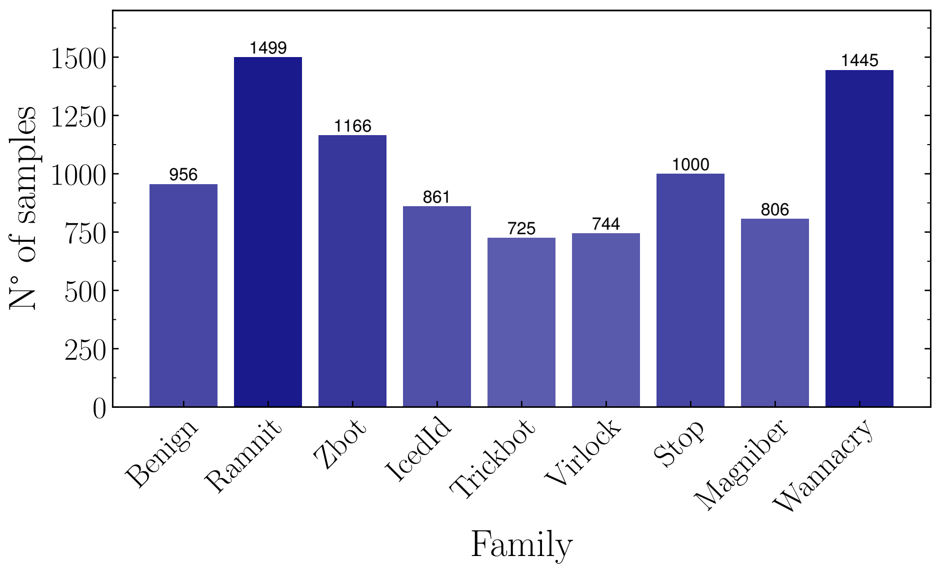


Figure 6.2. Number of samples per family for the *APIs* dataset.

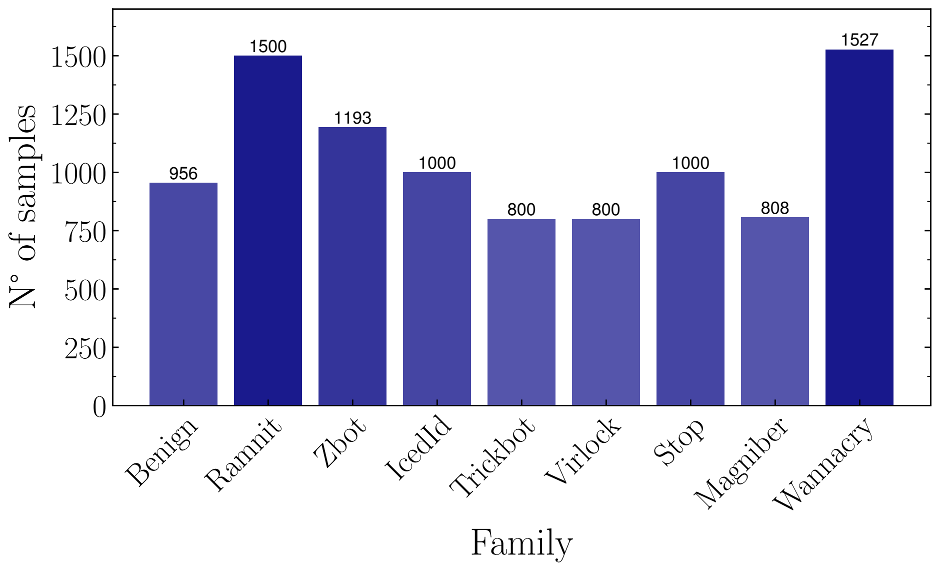


Figure 6.3. Number of samples per family for the *entropy* dataset.

Having to exclude some samples, these datasets required an additional revision in order to be combined. We prepared another dataset of combined features, where only those samples that have produced valid results for all the three selected features were included. Fig. 6.4 reports the number of sample for each class in this dataset.

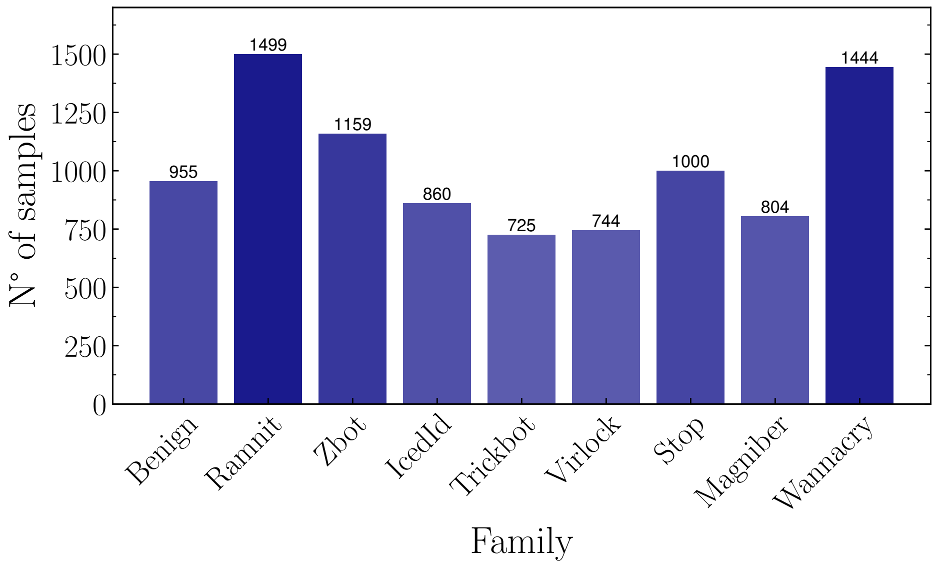


Figure 6.4. Number of samples per family when features are combined.

### 6.1.2 Dynamic features dataset

For what concerns dynamic features, only a smaller portion of the dataset has been subjected to dynamic analysis, since it was not possible to process the whole collection in an acceptable amount of time. For each class of the dataset, 200 samples were taken and analyzed.

## 6.2 Windows statistics

The following sections will give some insights on the data that has been extracted during the analysis of Windows samples, trying to emphasize differences and similarities that could be meaningful for both identification and characterization purposes. Each section is dedicated to one particular feature from the ones that have been obtained: *imported APIs*, *sections' entropy* and *dynamic APIs* (i.e. number of usages of APIs called by each sample).

### 6.2.1 Identification through APIs

Through our system, we have collected APIs contained in the Import Access Table (IAT) of each sample. We have counted, for each API, the number of times it appears in the IAT of benign and malicious samples, in order to compare it and check if this feature can be a good mean of identification. Figs. 6.5 and 6.6 depict, for benign and malicious samples respectively, the ten APIs that have been found most frequently. These two charts share some APIs, such as *GetLastError* and *Sleep*, but the majority of them differs, as well as the percentage values of the common ones. 82.2% of malware samples contain the *GetProcAddress* function, which can be used to retrieve the address of a precise procedure (i.e. a function) or a variable contained in a DLL. There are two other APIs that are meant to work with DLL: *GetModuleHandleA* and *LoadLibraryA*. Comparing the two charts, only four APIs out of twenty are shared, but with different percentage values.

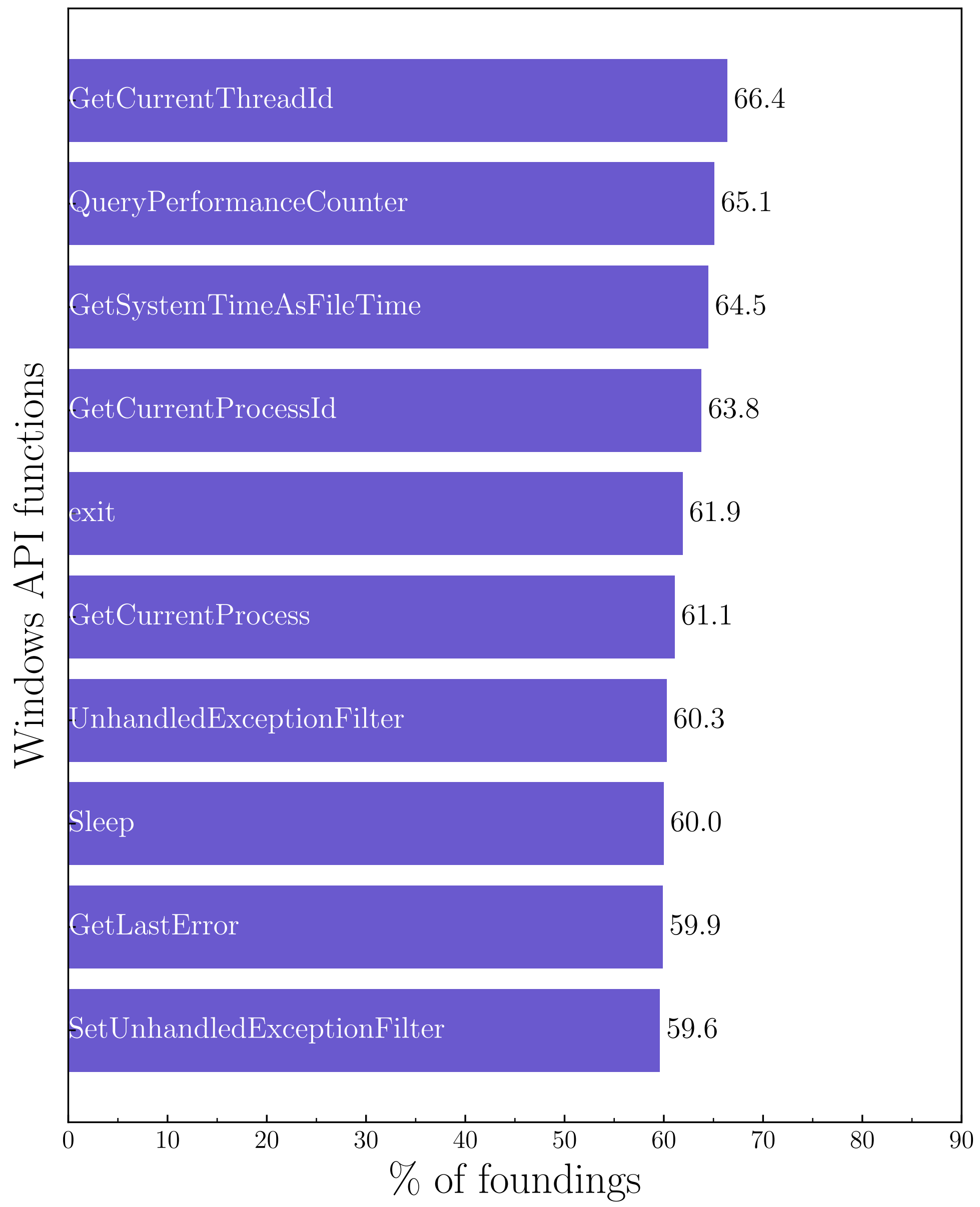


Figure 6.5. Number of appearances of APIs imported by benign executables.

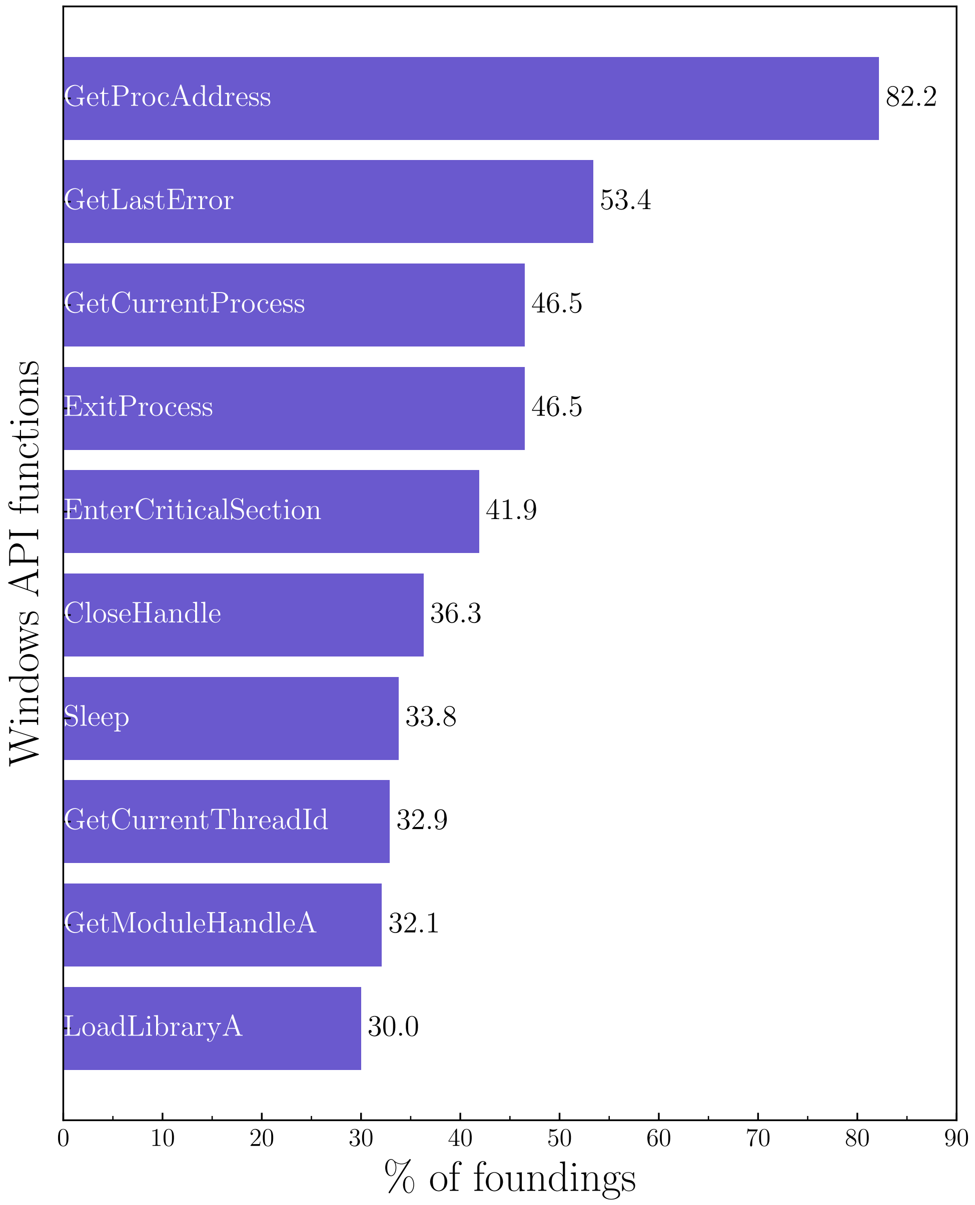
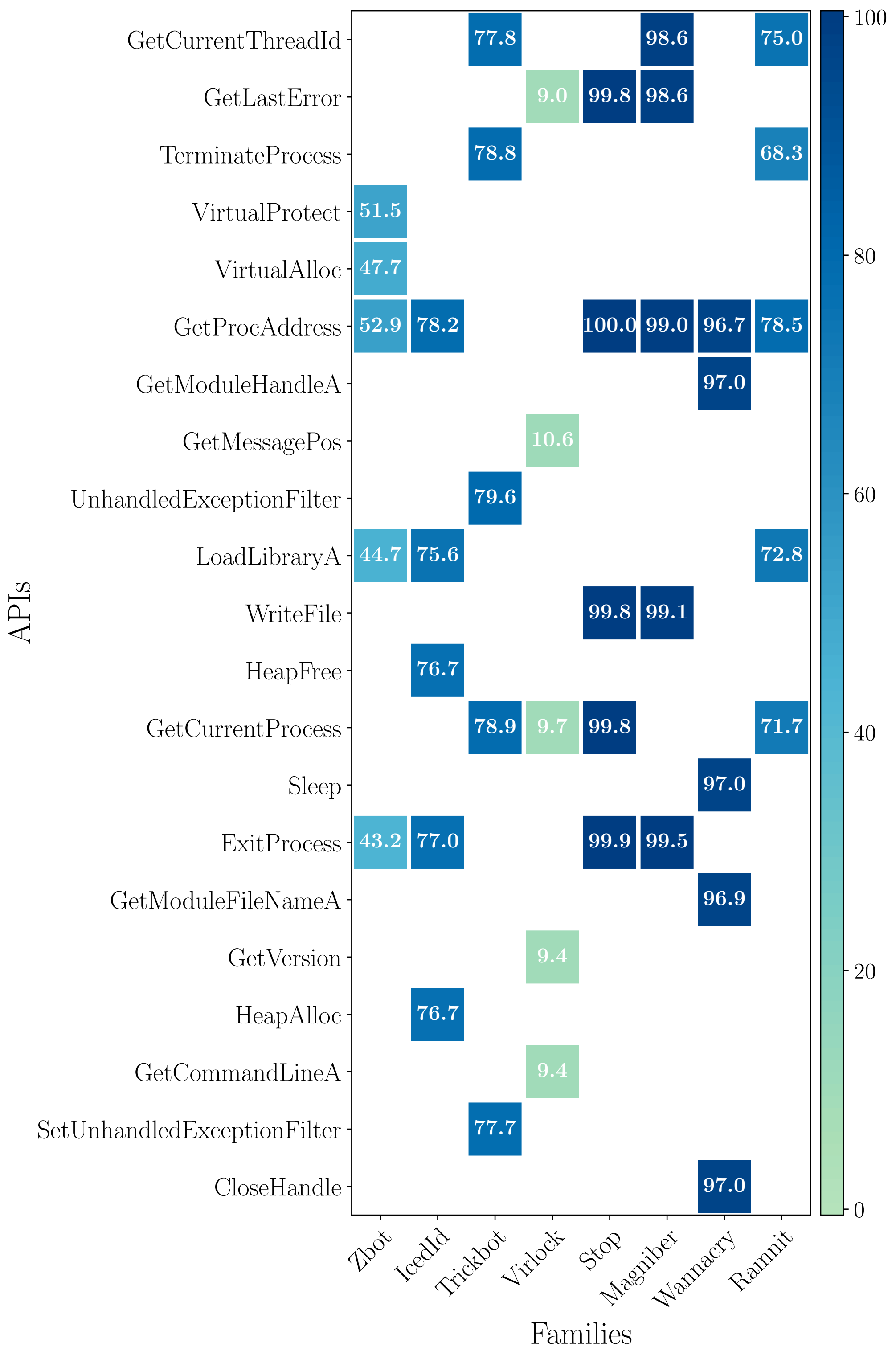


Figure 6.6. Number of appearances of APIs imported by malicious executables.

### 6.2.2 Characterization through APIs

Other than using APIs for identification, we would like to test this feature also for characterization. This section will present the distribution of some APIs on the various families we have collected, pointing out what are the main differences in this distribution.
We scanned the Import Access Table (IAT) of each sample counting, for each family, how many times each API has appeared. Then, we selected the five APIs that each family has mostly used and plotted all this information on the heat map depicted in Fig. 6.7. As can be noticed, most of the boxes are not present; this is due to the fact that the percentage of appearances was less than 1%. Despite we selected five APIs per family, the chart reports only 21 functions since some of them were common to more families. This map shows how some API functions are almost exclusive to a subset of the families, such as *TerminateProcess*, *LoadLibraryA* as well as some other such as *VirtualProtect, VirtualAlloc* are exclusive to a specific family. This is indeed a good sign that this feature could be used as a mean of characterization.



| APIs | Zbot | IcedId | Trickbot | Virlock | Stop | Magniber | Wannacry | Ramnit |
|---|---|---|---|---|---|---|---|---|
| GetCurrentThreadId | | | 77.8 | | | 98.6 | | 75.0 |
| GetLastError | | | | 9.0 | 99.8 | 98.6 | | |
| TerminateProcess | | | 78.8 | | | | | 68.3 |
| VirtualProtect | 51.5 | | | | | | | |
| VirtualAlloc | 47.7 | | | | | | | |
| GetProcAddress | 52.9 | 78.2 | | | 100.0 | 99.0 | 96.7 | 78.5 |
| GetModuleHandleA | | | | | | | 97.0 | |
| GetMessagePos | | | | 10.6 | | | | |
| UnhandledExceptionFilter | | | 79.6 | | | | | |
| LoadLibraryA | 44.7 | 75.6 | | | | | | 72.8 |
| WriteFile | | | | | 99.8 | 99.1 | | |
| HeapFree | | 76.7 | | | | | | |
| GetCurrentProcess | | | 78.9 | 9.7 | 99.8 | | | 71.7 |
| Sleep | | | | | | | 97.0 | |
| ExitProcess | 43.2 | 77.0 | | | 99.9 | 99.5 | | |
| GetModuleFileNameA | | | | | | | 96.9 | |
| GetVersion | | | | 9.4 | | | | |
| HeapAlloc | | 76.7 | | | | | | |
| GetCommandLineA | | | | 9.4 | | | | |
| SetUnhandledExceptionFilter | | | 77.7 | | | | | |
| CloseHandle | | | | | | | 97.0 | |

Figure 6.7. Number of appearances of APIs imported by malicious executables.

### 6.2.3 Identification through sections' entropy average

Let's now discuss another feature: *sections' entropy*. We extracted the entropy of every section contained in each executable, considering both standard and non-standard ones. The latter refers to those "custom" sections that are defined by the authors of the executable and they usually contain hidden data and code. Figure 6.8 reports the average of the entropy value for the standard section of benign and malicious samples. We did not included in this chart every possible standard section, since most of them were not likely to be found, hence we inserted only those that are usually more common. As can be seen from this chart, *.data, .idata, .pdata, .rsrc* and *.xdata* are the sections where the entropy values have a difference greater than one, while the other sections present values that are relatively close to each other.

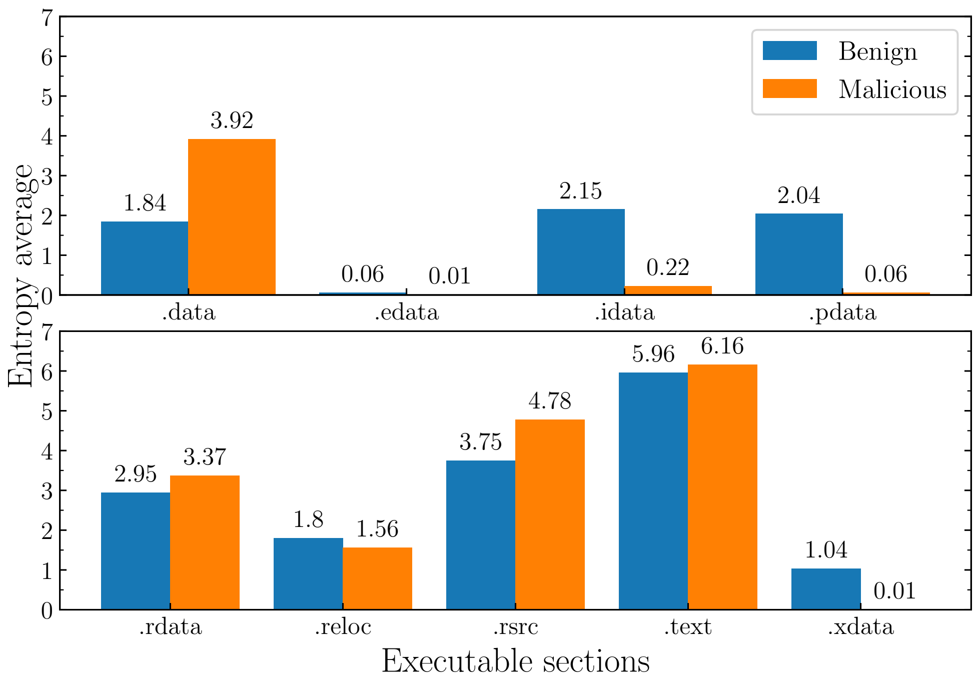


Figure 6.8. Sections' entropy average of malicious and benign executables.

## Number of "weird" sections

Other than considering just standard sections, we decided to take a look also to non-standard ones. We counted, for each family, how many times a "weird" section appears and calculated the average of this occurrence, which is shown in the bar chart depicted in Fig. 6.9. Ramnit family is the one that takes the lead, presenting almost two non-standard sections on each sample we have collected and examined, followed by Zbot with an average value of 0.79. On the opposite, Trickbot has very few samples that contain this kind of sections, showing an average value of 0.04 occurrences. IcedId, Virlock and Stop report similar values, ranging from 0.4 to 0.45 as well as Magniber and Wannacry with 0.19 and 0.23 respectively.

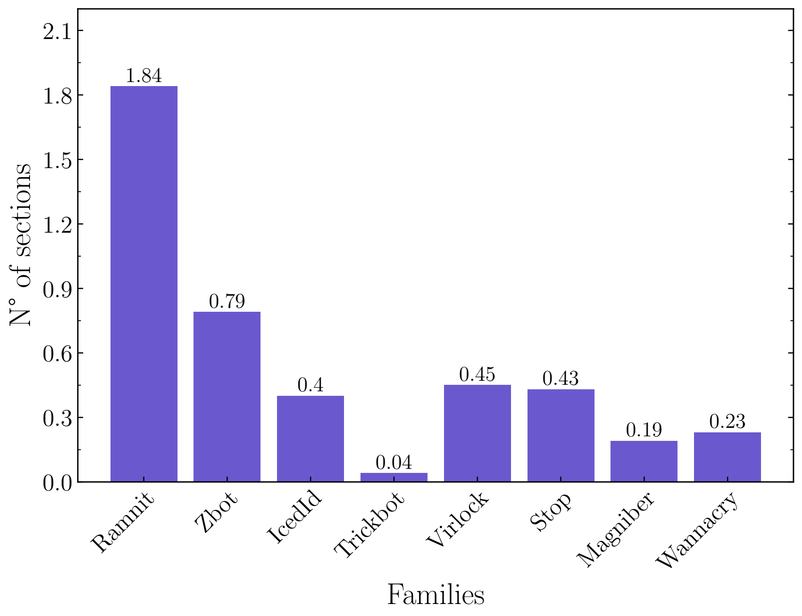


Figure 6.9. Average of weird sections found for each family.

### **"Weird" sections entropy average**

Other than considering just the number of times a non-standard section appears, we also decided to evaluate separately their entropy value. By looking at the bar chart depicted in Fig. 6.10, we can immediately notice that *Ramnit* has the highest entropy value which, compared to those of the standard sections (Fig. 6.8) is near to the *.text* section, which is usually meant for storing code. Other families, such as *Stop, Magnhiber and Wannacry* present lower values that are closer to *.rsrc, .data* and *.rdata*, which are standard sections meant for resources and data. The closeness of these values can be a good index of what these "weird" sections may contain, thus their content should always be considered when examining a sample.

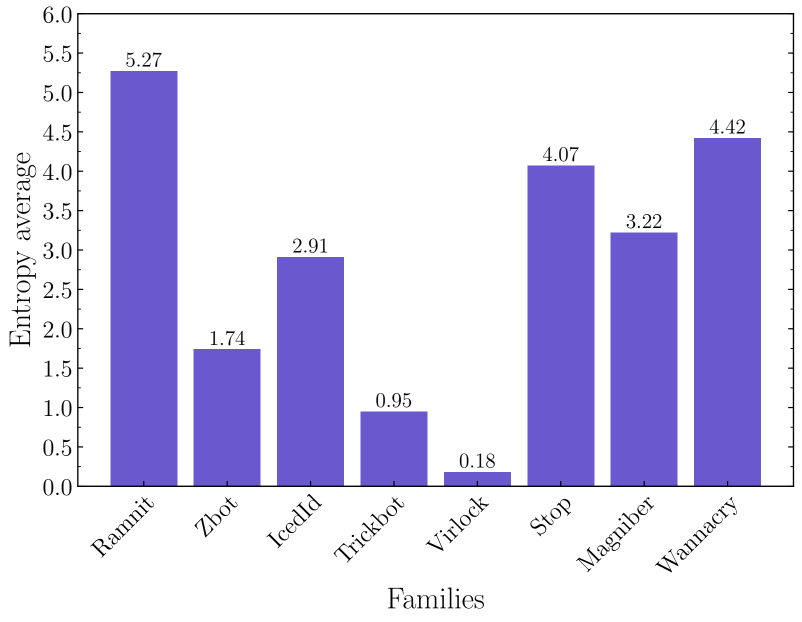


Figure 6.10. Average of weird sections' entropy found for each family.

### 6.2.4 Characterization through sections' entropy average.

Moving back to the entropy of standard sections, let's now discuss these values family by family. The map depicted in Fig. 6.11, shows the entropy average of each standard section for each family. White boxes represent value that had an average value between 0.00 and 0.01. When calculating the average and a standard section did not appear in the executable, we assigned it the value 0.0. By looking at this heat map we can immediately see how all the families but Zbot have high values of entropy on the *.text* section, ranging from 6.14 (Wannacry) to 7.88 (Virlock). Sections *.edata, .idata, .pdata* and *.xdata* seem to have not meaningful differences in their values, since each family has low entropy that stands between 0.0 and 0.25, except for Trickbot which has a slightly higher value on the *.idata section* (1.0). The remaining sections definitely show higher values and in each of them there is at least one family that distinguishes: Zbot has the lowest rate in each of these sections except for *.rsrc* and *.reloc*.
These results show how not every standard section has to be considered and probably, only *.data, .rdata, .reloc, .rsrc* and *.text* would be enough for characterization purposes. Tab 6.1 summarizes, for each of these sections, what is the family that has the highest and the lowest value. It clearly shows how in these five sections all the families except Trickbot distinguishes itself for having the highest or the lowest value.

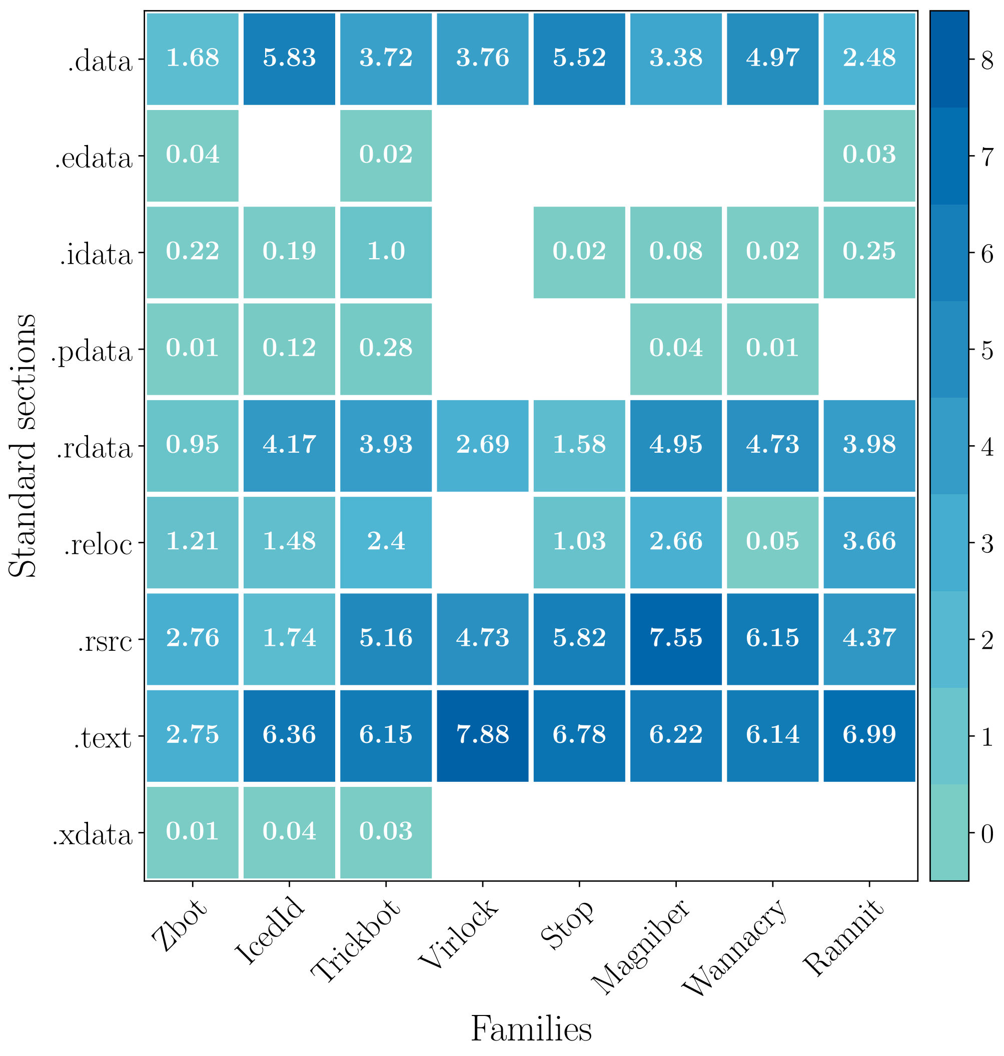


Figure 6.11. Average of sections' entropy for each family.

| Section | Highest entropy family | Lowest entropy family |
|---|---|---|
| *.data* | IcedId | Zbot |
| *.rdata* | Magniber | Zbot |
| *.reloc* | Ramnit | Wannacry |
| *.rsrc* | Magniber | IcedId |
| *.text* | Virlock | Zbot |

Table 6.1. Sum up of the most meaningful sections.

### 6.2.5 Identification through dynamic APIs

After having considered the previous **static** features, it is time to move onto the dynamic one: *API counts*. These have been captured by executing 200 samples of each family inside a sandbox and monitoring each call they made. This section will discuss identification, by comparing the APIs that were called mostly by benign and malicious samples respectively. Fig. 6.12 reports the ten top APIs for benign executable, while fig. 6.13 shows the malicious ones. Both charts have *NtClose* and *LdrGetProcedureAddress* as the two most used APIs but the real interesting difference is in the number of calls, and this looks huge. Despite both benign and malicious samples were ran for five minutes, malicious ones present a more intense usage of the system capabilities, which could probably be considered as a stand-alone feature, without actually considering which are the functions that have been actually called.

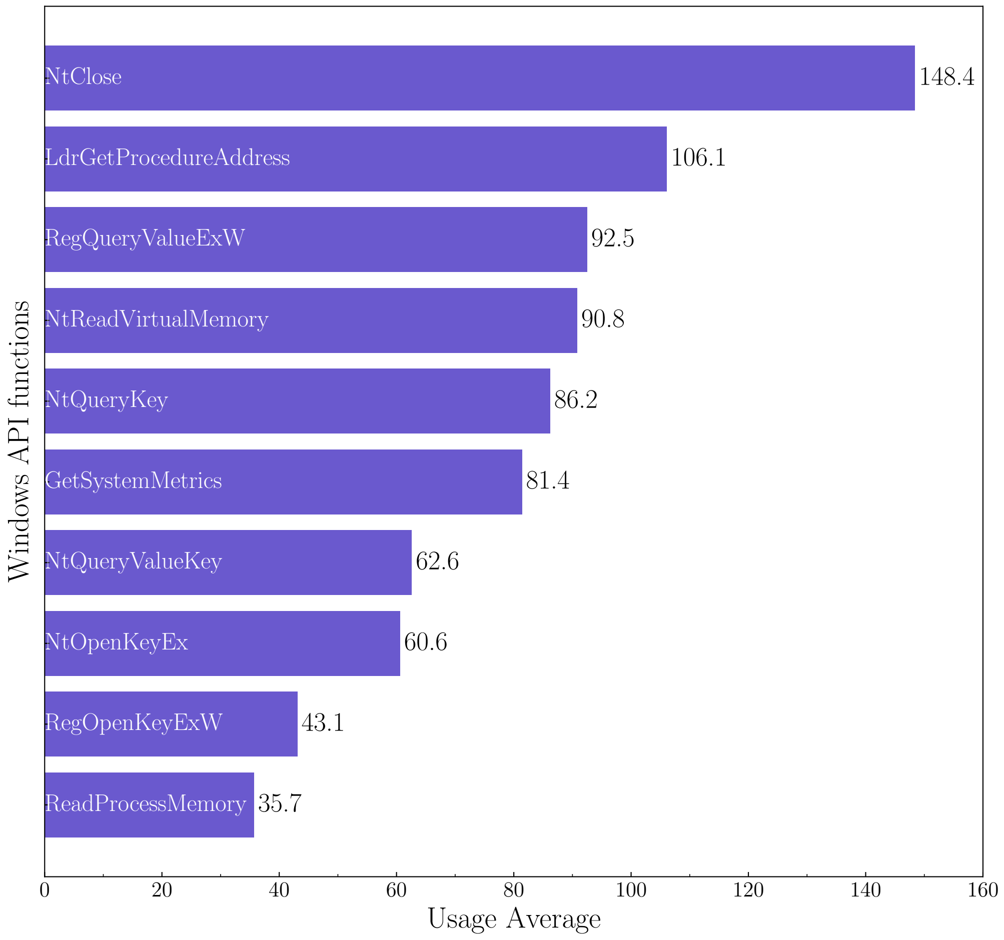


Figure 6.12. Average of number of calls for benign samples.

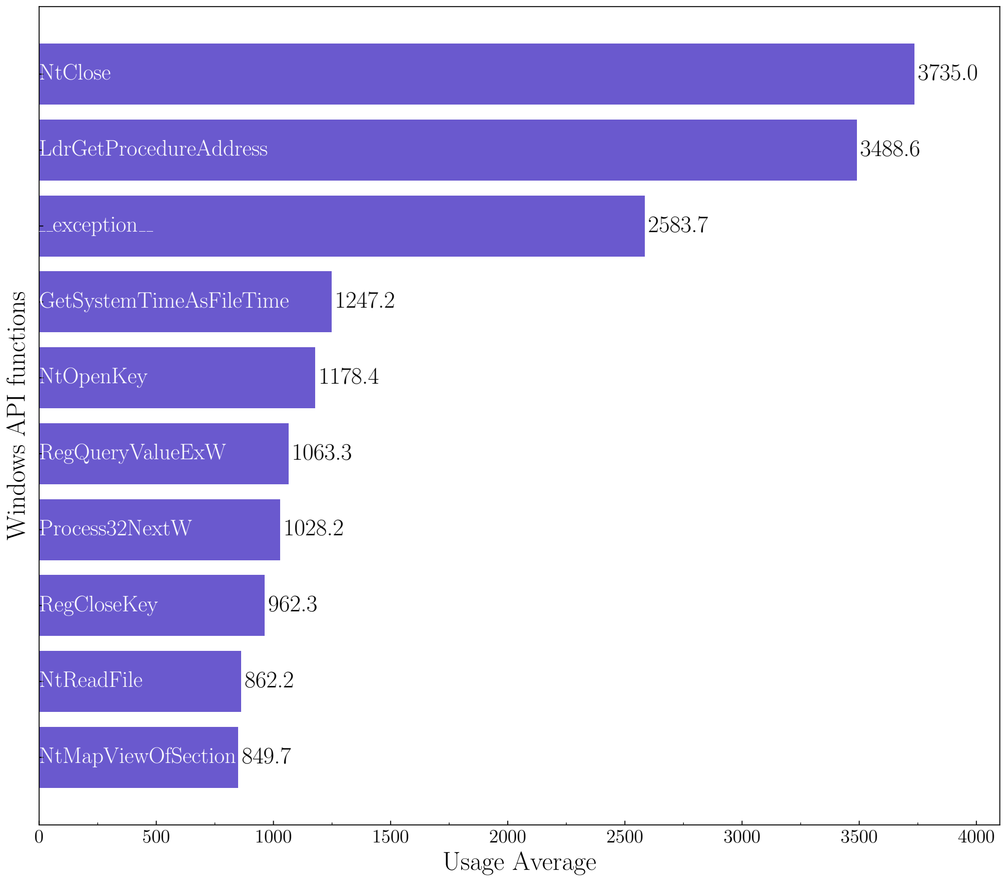


Figure 6.13. Average of number of calls for malicious samples.

### 6.2.6 Characterization through dynamic APIs

After having discussed those APIs that are mostly used by malicious samples, it is now time to take a deeper look on what each family actually does. Figs. 6.14, 6.15 show the average number of calls performed by each family for a subgroup of APIs. These two charts report a total of 28 functions, which have been selected by grabbing the top five used APIs of each family. These two maps reveal that the numbers reported in the previous section, in Fig. 6.13, do not describe correctly the behavior of some families. The two APIs that appeared before as the mostly used, are actually heavily used by only two families: Virlock and Stop. Among our families, Virlock is the one that has performed the highest number of function calls, reporting the highest values in half of the selected functions. On the contrary, other families seem to have only a few functions were they take the lead in the number of calls. The first map, depicted in Fig. 6.14, also shows that some APIs were not even found on every family, such as *FindResourceExA, SizeOfResources* and *OutputDebugString.A*, with the last one that has appeared only on Zbot samples.

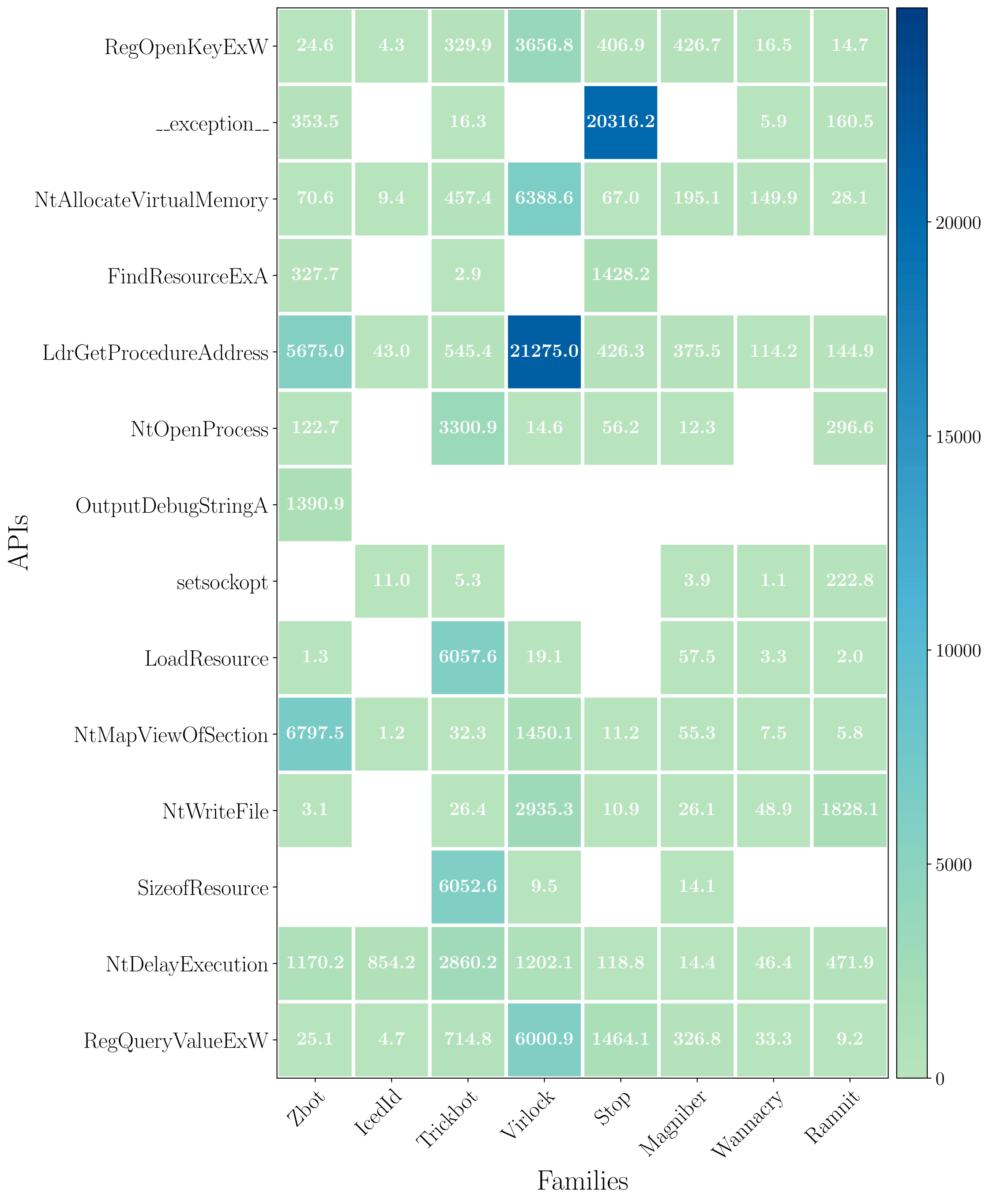


Figure 6.14. Average of number of calls for benign samples.

The second chart, depicted in Fig. 6.15, shows a more homogeneous situation, since all the 14 APIs are used by at least six families out of eight, but with substantial differences in the number of calls performed. As already said, Virlock presents the highest number in the majority of these function calls, but we can find interesting values in other families as well. For example, Ramnit samples seem to have extensively used the *ReadProcessMemory* function, showing an average of 4109.3 calls, which is quite an outstanding value not only in comparison to other families, but even when comparing to other API calls of this family. Similar situations occur for other families as well: IcedId with *NtDelayExecution* (854.2 calls), Zbot with *LdrGetProcedureAddress* and *NtMapViewOfSections* (5675 and 6797.5 calls respectively).

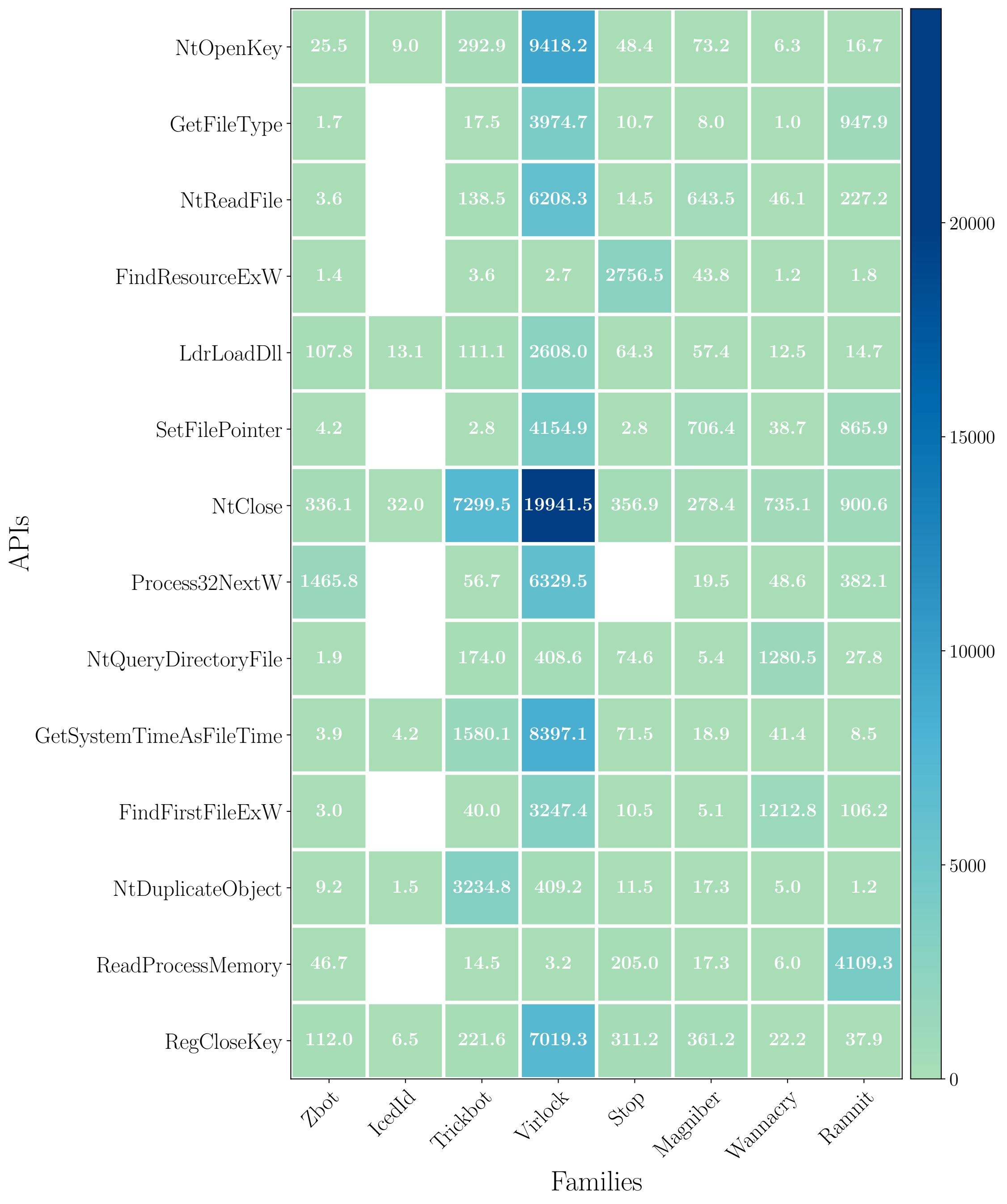


Figure 6.15. Average of number of calls for malicious samples.

## 6.3 Android Statistics

This section will describe, in the same manner as for the Windows part, the features that have been extracted from Android samples. Differently from before, the following sections will just discuss and describe these results for characterization purposes, not identification, since the Android collection does not include a benign class, but only seven malware families. As already discussed in chapter four, the Android collection is not well-balanced as the Windows' one, since we were not able to find a substantial number of samples for some families, but we though their results still deserve to be discussed and reviewed.

### 6.3.1 Characterization through permissions

The permission that an application requires in order to run, can often portray very clearly what it is going to perform, or at least, what it will be capable to perform. There are many kinds of permission in the Android ecosystem. While analyzing our samples we came across 253 different permissions,. Some of them were present almost in each analyzed sample, while some others rarely appear and only a small subset of sample requires them. This section will report some statistics about some of this permissions, covering only 15 out of the 253 found. These 15 samples were selected mainly for two reasons:

1. Percentage of samples containing them. This should be greater than 1%.

2. Potential danger of such permissions. For example the *SYSTEM_ALERT_WINDOW* can be required to perform an overlay attack. Some other permissions such as *READ_SMS* and *PROCESS_OUTGOING_CALLS* can represent a menace to user's privacy.

Tables 6.2, 6.3 and 6.4 describes the permission reported in the three charts below. These description are provided by the Android Developer Manual [104].

| **Permission** | **Description** |
|---|---|
| RECEIVE_BOOT_COMPLETED | Allows an application to start at each boot. |
| ACCESS_FINE_LOCATION | Allows an app to access precise location. |
| ACCESS_COARSE_LOCATION | Allows an app to access approximate location. |
| READ_SMS | Allows an application to read SMS messages. |
| SEND_SMS | Allows an application to send SMS messages. |

Table 6.2. Description of some of the selected android permissions.

| **Permission** | **Description** |
|---|---|
| PROCESS_OUTGOING_CALLS | Allows an application to see the number being dialed during an outgoing call with the option to redirect the call to a different number or abort the call altogether. |
| CALL_PHONE | Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call. |
| READ_PHONE_STATE | Allows read only access to phone state, including the current cellular network information and the status of any ongoing calls. |
| KILL_BACKGROUND_PROCESSES | Allows an application to request an install killing of one or more processes. |
| WRITE_SETTINGS | Allows an application to write the system settings. |

Table 6.3. Description of some of the selected android permissions.

| Permission | Description |
|---|---|
| RECORD_AUDIO | Allows an application to record audio. |
| READ_CONTACTS | Allows an application to read the user's contacts data. |
| CAMERA | Required to be able to access the camera device. |
| DISABLE_KEYGUARD | Allows applications to disable the keyguard. |
| SYSTEM_ALERT_WINDOW | Allows an app to create windows on top of all other apps. |

Table 6.4. Description of some of the selected android permissions.

Figures 6.16, 6.17 and 6.18 portrait, for each family, the percentage of samples that contain these permissions in their Manifest. These charts show interesting results, since:

- There is a lack of some permissions in some families. For example, Ewind samples do not have any of the permissions listed in Tab. 6.4, as can be seen in Fig. 6.18. In the first chart depicted in Fig. 6.16 can be seen how Svpeng does not have neither *ACCESS_FINE_LOCATION* nor *ACCESS_COARSE_LOCATION*. In Fig. 6.18 Anubis does not show any tracks of *PROCESS_OUTGOING_CALLS* and *WRITE_SETTINGS* permissions.
- Some permissions are mainly used by a single family: we can notice in the chart of Fig. 6.18 that the 60% of Spveng samples require the permission *DISABLE_KEYGUARD*, while all the other families do not exceed the 10%. Another example is in Fig. 6.17 where the permissionn *WRITE_SETTINGS* is mainly used, again, by Svpeng for which almost 90% of samples have it, followed by HiddenAd with a value close to 20%.

There are also some permissions there are very common between the selected families, such as: *RECEIVE_BOOT_COMLPETED* and *READ_PHONE_STATE*.

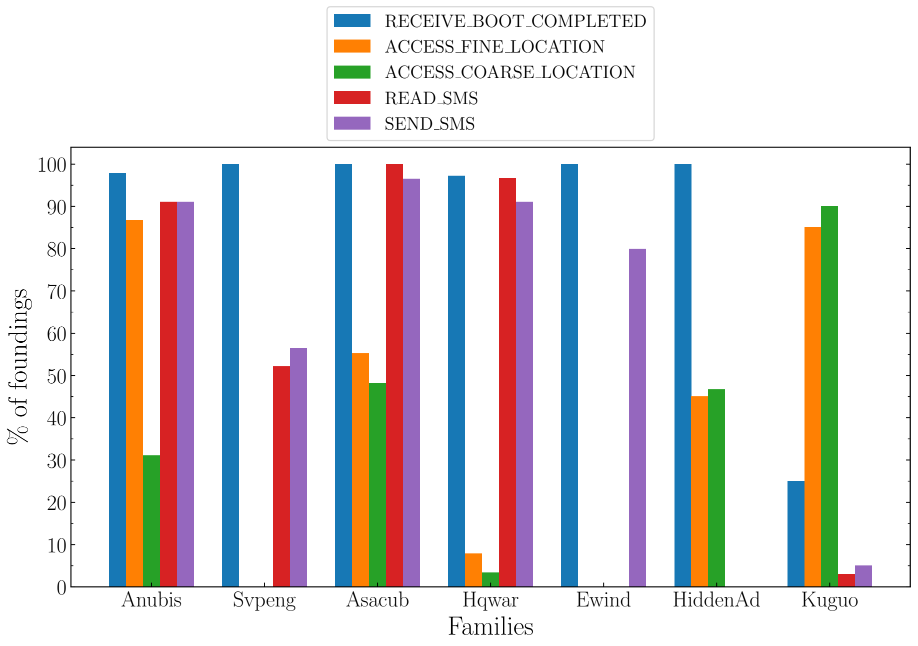


Figure 6.16. Percentage of samples containing some specific permissions.

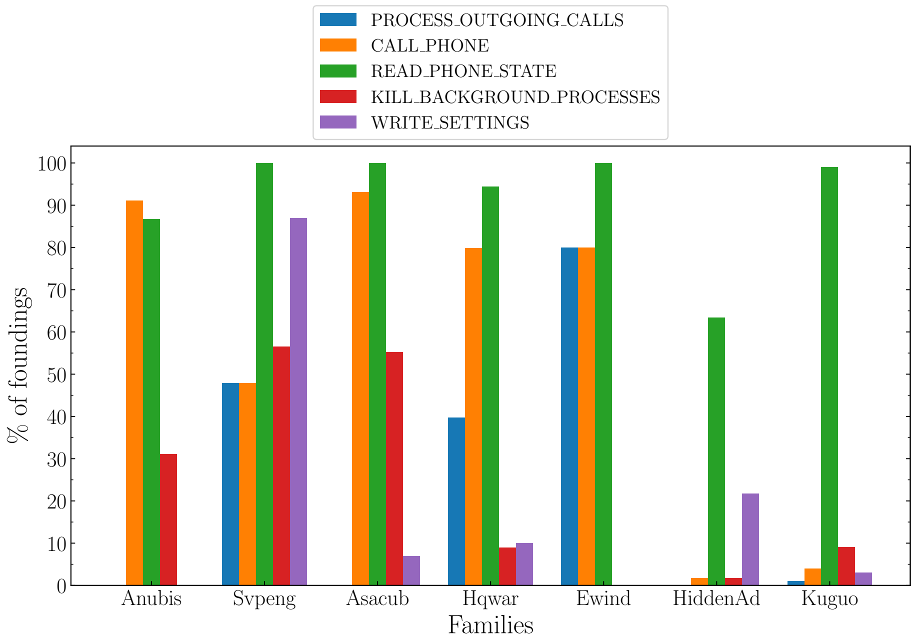


Figure 6.17. Percentage of samples containing some specific permissions.

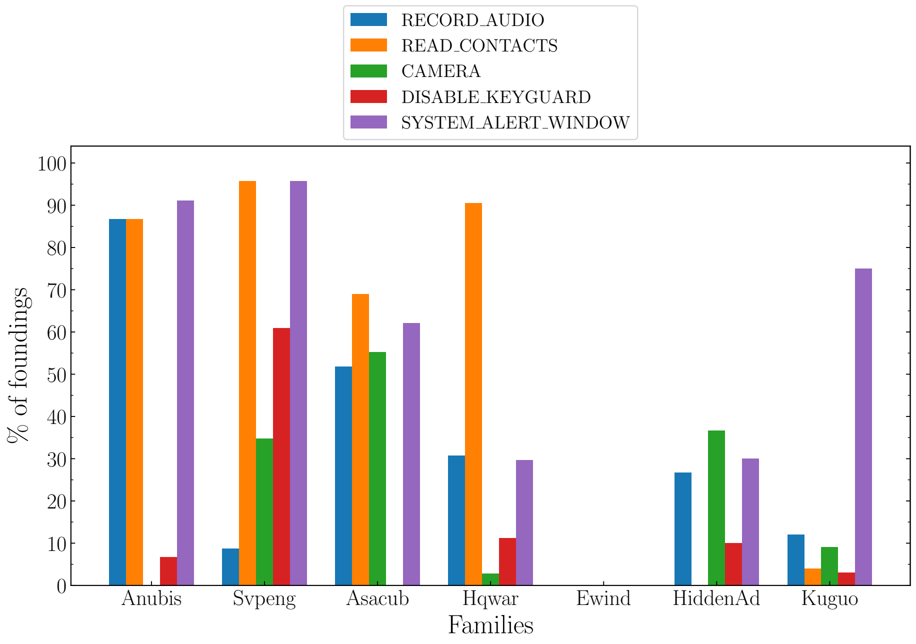


Figure 6.18. Percentage of samples containing some specific permissions.

### 6.3.2 Characterization through code properties

Using Droidlysis [99], we extracted also the application code properties. These properties, similarly to permissions, gives a general description of what the application's code does. Once again, we did a selection among the properties encountered, since some of them did not provide more information than permissions, so this section will report results meant to describe different kinds of behaviors. Tables 6.5, 6.6 describe the selected code properties. These description are provided by the Droidlysis documentation [99].

| **Property** | **Description** |
|---|---|
| Obfuscation | Obvious traces of code obfuscation. |
| Accessibility_services | Work with accessibility settings (use, or implement a service). |
| Encryption | Uses encryption. |
| Set_component | Might be trying to hide the application icon. |
| Tasks | Lists running tasks. |

Table 6.5. Some of the selected code properties.

| **Property** | **Description** |
|---|---|
| Reflection | Uses Java Reflection. |
| Device_admin | Creates or uses a device administrator app. |
| Base64 | Uses Base64 encoder/decoder. |
| Stacktrace | Get stack traces. Can be used as Anti Frida technique. |
| Execute_native | Executes shell or native executables. |

Table 6.6. Some of the selected code properties.

The two charts below, depicted in Figs. 6.19, 6.20 report, for each one of the selected properties, the percentage of samples of each family that have that property. As can be seen, there are some which are shared, with different percentages, between families: *obfuscation, accessibility services, set component, tasks, reflection* and *stacktrace*. Kuguo seems to be the one that often uses obfuscation techniques, presenting the highest value for the *obfuscation* property as well as the *execute native* which can be an index of hidden payloads that are lately loaded by the malware. Asacub appears to be the only family that never uses encryption. All these families have at least the 20% of their samples that try to install an accessibility service, which as already discussed before, is a quite popular technique employed by malware to perform malicious actions.

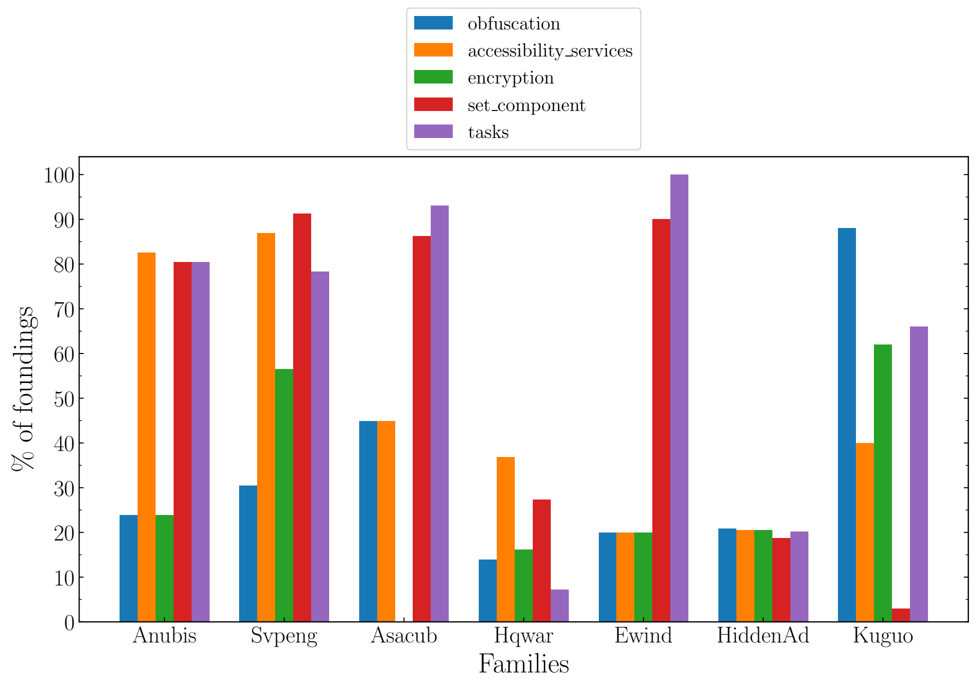


Figure 6.19. Percentage of samples containing specific smali properties.

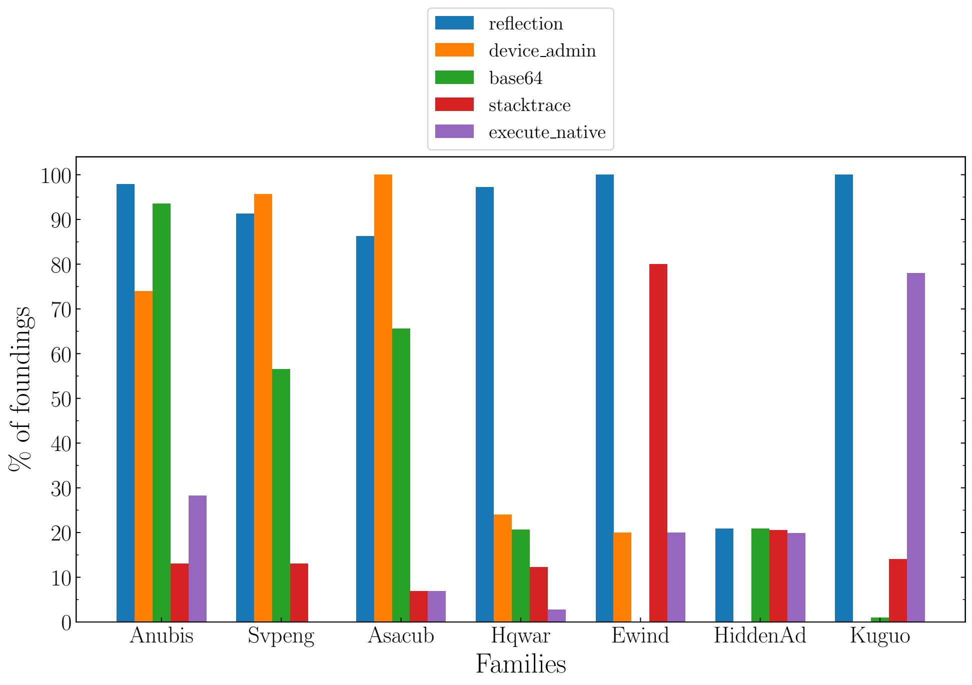


Figure 6.20. Percentage of samples containing specific smali properties.

### 6.3.3 Considerations on the "Activities" feature

Among the selected feature for Android samples, we also considered, in the first place, the names of the Activities declared by each analyzed application. Unfortunately, we had to discard this feature since these activities were often declared with random or encrypted/encoded names, hence there were no relations between samples of the same families and using just their name would be useless for Machine Learning purposes. They should probably be taken in consideration in a different way, not grabbing just their name but taking a deeper look on what they contain.

### 6.3.4 Characterization through dynamic APIs

Android samples have been dynamically analyzed as well, executing them inside an emulator and monitoring them through CuckooDroid [72]. Among the various information that CuckooDroid is able to provide, we selected the API count (i.e. number of times that each function is actually called). Each malware sample has been ran for five minutes on the emulator. Unfortunately, Cuckoo is not able to interact with an emulator that runs a modern version of Android OS and we had to emulate a device with Android 7.0 on it. The main drawback of this limitation is that if an application requires a newer OS version, it will not be ran and the analysis will fail. Luckily, the majority of the samples we collected worked with no troubles. Figure 6.21 illustrates the average of the number of calls for some APIs done by each family. Overall, we encountered 65 different APIs across different samples. The ones reported in this chart have been selected by taking the 5 functions that were mostly called by each family. The first two values that can immediately be noticed are those of Anubis and Asacub on the two APIs Base64.decode and ActivityManager.getRunningTasks. These two functions are intensely used, reporting values that are significantly higher than those reported by other families.

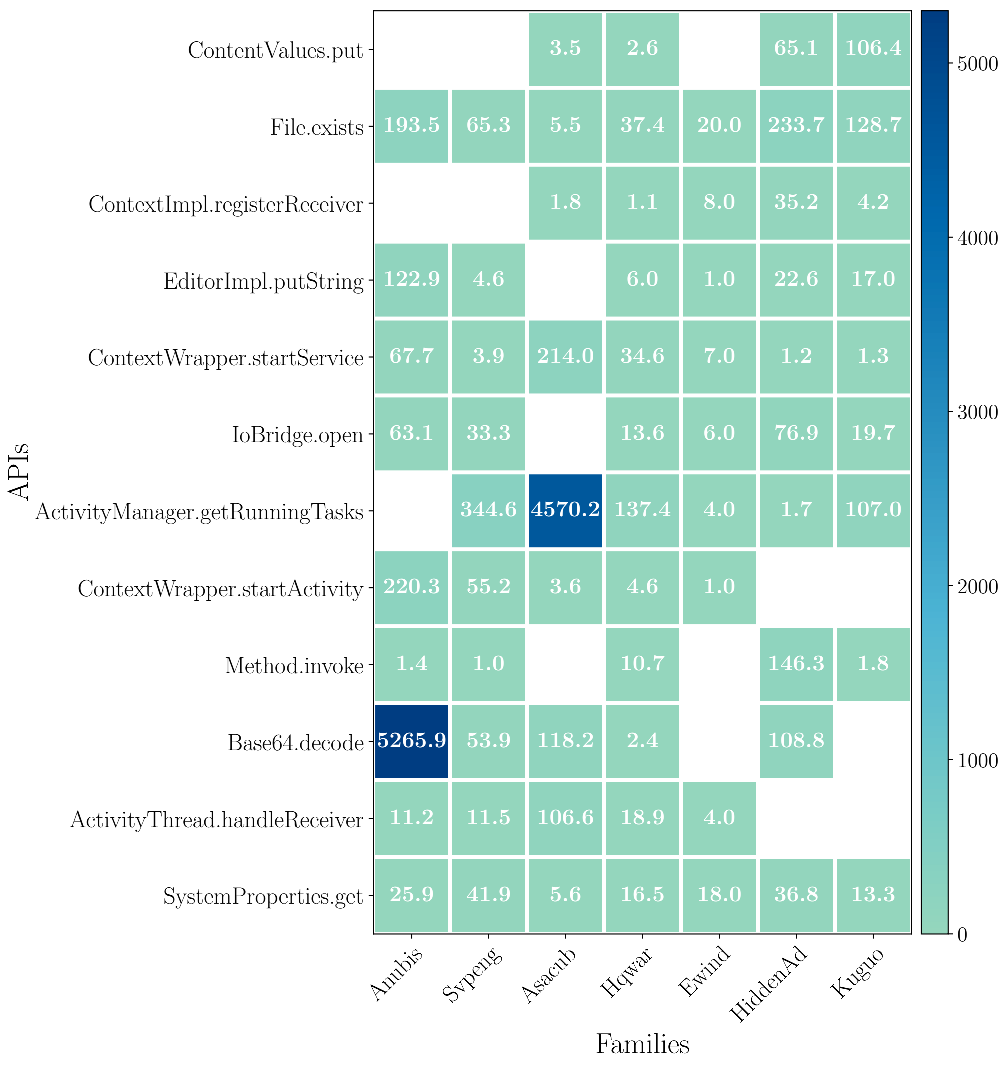


Figure 6.21. Average of number of calls for malicious samples.

Ewind samples are the ones which performed the lowest number of function calls, since their values range from 0 to 20 (File.exists). All the other families seem to have at least one or two functions that have used the most. For example, most of the calls of Svpeng and Hqwar are on the ActivityManager.getRunningTasks, while Kuguo concentrates its calls on ContentValues.put, File.exists and ActivityManager.getRunningTasks with 106.4, 128.7 and 107.7 calls respectively. According to these values, this feature can provide a good representation of the family's behavior, since it is possible to outline differences.

# Chapter 7

# Conclusions

The main objective of this work was to define a methodology for malware analysis, in order to outline what are those factors that could play a significant role in the characterization process. This was done by designing an analysis system which, applying both static and dynamic techniques, is able to extract meaningful information, also called features, from samples. The extracted information has later been packed and prepared to be served as training input for a Machine Learning (ML) system.

In order to achieve this task, we started from a first outline of the research scopes (i.e. which Operating System to cover, which types of malware etc.). This was done by collecting as much information as possible on the current malware landscape, to gain knowledge on which families are more common nowadays as well as looking at the state-of-the-art of malware analysis techniques and tools. Once these scopes were defined, we moved to a more deep and technical study of malware behaviors by collecting technical analyses of samples belonging to a selection of families, in order to gain more knowledge on how the analysis process can be carried on and what are the tools and techniques that are commonly employed.

Before to actually develop our analysis system, we needed to harvest some samples. To this end, we used different malware sources, combining them together to create our own collections. Despite there were already some collection available in the wild, we wanted to improve some aspects, such as having a balanced number of samples per class (i.e. family) and including only those families that are, at least in these days, common.

Using all the knowledge gained in the previous steps, we started building our own analysis system, made of a static and a dynamic analyzer capable of extracting some types of features from the two collections mentioned above. We designed this system starting from some well-known analysis tools, such as the Cuckoo sandbox [69], automating their functionalities in order to apply them to a large collection. As results of our analysis, we created a series of datasets containing only wanted features, which have been described and reviewed in Chapter 6.

These features have been carefully reviewed, in order to check if they actually revealed some differences in the behavior of the selected families. Overall, every feature from the ones we selected showed significant differences between samples belonging to different families. Looking at the Windows environment, the entropy of non-standard sections and the number of API calls seem to be the two features that mostly characterize each family, but the latter requires definitely more time to be collected, since samples have to be ran and monitored. On the other side, the features extracted from Android samples showed interesting differences on the kind of actions that each family tries to perform. We tried to focus on those actions that can actually be harmful, such as permissions and code properties that are required to perform specific attacks or to access to privileged data. Among these features, we had to discard the Activities' names, since they were often random or encoded in some manner, being not adequate for our purposes.

## 7.1 Future works

The analysis system developed, could potentially be of great help if we want to extract representative information from a batch of samples. Our system could be extended, adding more types of features that will be extracted or integrating other tools to analyze samples. There are two main limitations that we recognize:

1. **Supported file formats**: these should be definitely extended, since malware may come in many different flavours (i.e. office documents, pdf etc.). Supporting different kind of file types would also make easier to create a suitable malware collection to test, since when we created our two collections we found many samples which where in different formats (mainly dlls for Windows, and dex files for Android).
2. **Extracted features**: the features we were able to extract are quite general and simple. These could be extended by integrating, for example, more specific aspects (i.e. look for APIs that perform access to the Windows registry only instead of each possible API). There were many features that we thought of, but did not have the time to integrate, such as the dimension of non-standard sections or the system calls done by Android APKs.

For what concerns dynamic analysis, these definitely require a substantial amount of time and resources in order to be performed. Cuckoo offers the possibility to configure a distributed environment, which would be a nice feature to integrate in our system, adding the chance to apply dynamic analysis to a larger amount of samples, in a feasible amount of time.

# Appendix A

# User manual

Our analysis system consists of two main parts: a *Windows* and an *Android* analyzer. These require the installation and configuration of some tools in order to run. The code repository consists of the following files:

- analyzer.py: this is the core of the analyzer, it contains a series of functions that are in charge of grabbing samples and process them, extracting wanted features and packing them.
- android_static.py: it contains the code needed for the Android static analyzer.
- clientAPI.py: it contains functions that interact with Cuckoo's API component.
- conf.py: this file needs to be modified to adapt it to your evenironment and customize the analysis.
- extractFeatures.py: it retrieves features from the results produced during the analysis.
- mysecrets.py: it contains the Cuckoo API's key that is needed in order to interact with the sandbox. This is actually not contained in the repository, should be added manually.
- requirements.txt: list of libraries that are required to run the system.
- cdrequirements.txt: list of CuckooDroid's required libraries.

Once the repository has been downloaded, the required libraries can be installed with the following command:

Listing A.1. Installing required libraries.

```
pip install -r requirements.txt
```

This command installs some basic libraries, however there are some more complicated components that require some steps in order to be properly configured. These will be explained in the following sections.

## A.1 Configuring the Android Static analyzer

The static component of the Android analyzer only uses DroidLysis to perform analysis. Luckily it is pretty straight forward to configure by following the instruction on its Github page.

## A.2 Installing and configuring the Cuckoo sandbox

Cuckoo provides and automatic sandbox, that once a sample is submitted, it will be sent to a Virtual Machine and ran, monitoring various kind of actions. It can be installed following its Documentation, however we faced some troubles by following the standard configuration, so this section will provide all the useful steps that we took to make it work properly with our system. Cuckoo requires many prerequisites before actually starting using it. First of all, we need to install some libraries, with the following commands:

Listing A.2. Installing Cuckoo requirements.

```
sudo apt-get install python python-pip python-dev libffi-dev libssl-dev
sudo apt-get install python-virtualenv python-setuptools
sudo apt-get install libjpeg-dev zlib1g-dev swig
sudo apt-get install mongodb
sudo apt-get install tcpdump apparmor-utils
sudo aa-disable /usr/sbin/tcpdump
sudo groupadd pcap
sudo usermod -a -G pcap cuckoo
sudo chgrp pcap /usr/sbin/tcpdump
sudo setcap cap_net_raw,cap_net_admin=eip /usr/sbin/tcpdump
sudo apt-get install swig
```

Once all these package are installed we can move to download and install Cuckoo. It is suggested to install its Python dependacies inside a virtual environment. This can be done by running:

Listing A.3. Installing Cuckoo requirements.

```
sudo adduser cuckoo
sudo usermod -a -G vboxusers cuckoo
virtualenv venv
. venv/bin/activate
pip install -U pip setuptools
pip install -U cuckoo
```

After these commands we need to run Cuckoo to make it prepare its working directory, which will be placed in our home directory. Cuckoo can be launched with:

```
cuckoo -d
```

### A.2.1 Configuring the Virtual Machine

It is now time to configure the guest component, which is the actual sandbox where samples will be executed. We used VirtualBox to set up this Virtual Machine, creating a VM with the specifications reported in Tab. A.2.1

| **Windows 7 VM** | |
|---|---|
| Name | cuckoo1-win7 |
| Guest OS | Windows 7 (64 bit) |
| Memory size | 4096 MB |
| Number of CPUs | 2 |
| Shared folders | None |
| Network type | Host-only |

Table A.1. Configuration information of the Windows 7 Virtual Machine.

Once the VM is configured and ready to be used, we can start customizing it to make it interact with Cuckoo. First of all we need to install Python on it. During this customization

part, we will set the Network to *NAT* mode, enabling Internet access. Python can be installed from the official website.
Then, we need to modify Windows settings to disable both Firewall and Automatic updates, since these two may affect samples behavior. This can be done from the control panel. Now we can set the network back to *Host-only* mode, which needs an adapter named *vboxnet0* (default name). The adapter can be created from VirtualBox from File-Host Network Manager.

### A.2.2 Configuring the network

Cuckoo offers various types of network modes, from complete isolation to full internet access (which is obviously not suggested). Since there are some malware that will not run if access to the internet is not provided, we decided to use the InetSim routing, which simulates some network services, hoping that this will be enough to trick our samples.
InetSim can be installed from its main page. It requires Perl, which can be obtained by running:

```
sudo apt install perl
```

It also requires some specific libraries, these are listed on the website. In order to install a library, it has to be downloaded and installed from the main website (the previous link provides pointer to those libraries), unpacked and then, once inside its folder:

```
perl Makefile.pl
make
make test
make install
```

Now we just need to configure InetSim and the Windows VM to work together. We need to modify InetSim's configuration file, named *inetsim.conf*, which can be found inside the *conf* folder. This file needs to be modified adding the following lines:

```
service_run_as_user cuckoo
dns_default_ip 192.168.56.1
```

For what concerns the Windows7 VM, we need to configure the DNS address. This can be set from the control panel, as explained by this guide. The DNS address must be *192.168.56.1*, which is the one inserted in the InetSim configuration file. Then we just need to run, from the inetsim folder:

```
sudo ./inetsim
```

### A.2.3 Final steps

Once all the prerequisites have been installed, we just need to configure Cuckoo before running it. The official documentation clearly explains how these should be modified. We followed the standard configuration, changing only the routing mode in *routing.conf* to select InetSim as routing option. After filling those files, we need to move back to the VM, take the file *agent.py* contained in the Cuckoo folder and run it with Python. The file can be moved to the VM with shared folders or downloading it from Github (just remember to disable shared folders or to switch back to *Host-only* mode). Then, we take a snapshot of the VM, running on a terminal:

```
VBoxManage snapshot "<Name␣of␣VM>" take "<Name␣of␣snapshot>" --pause
VBoxManage controlvm "<Name␣of␣VM>" poweroff
VBoxManage snapshot "<Name␣of␣VM>" restorecurrent
```

In our case, the VM was named *cuckoo1-win7* and the snapshot *clean*. Now Cuckoo should be correctly configured and ready to be ran.

### A.2.4 Some possible caveats

During our experience with Cuckoo we faced some problems that are not covered by the standard documentations. This section will contain some solutions to possible troubles an user may face while using our system.

- Cuckoo offers a Web interface to interact with it, unfortunately we did not manage to make it work on modern versions of Ubuntu (i.e. 20-22) but only on older ones, such as the 17.
- Cuckoo API did not work at first. It requires a precise version of a Werkzeug (a Python package). This can be installed by running:

  ```
  pip install werkzeug==0.16.1
  ```

- Always start Inetsim before Cuckoo.
- Inetsim will not work if an address is not assigned to the *vboxnet0* interface, which may happen if you have not open a VM after the boot of your system. You can assign an address manually with the command:

  ```
  sudo ifconfig vboxnet0 192.158.56.1/24
  ```

- Always take VM snapshots using the command line, never from the VirtualBox GUI.

However, for any problem not mentioned here, there is an issue page that contains many possible problems with some solutions.

### A.2.5 Running Cuckoo

Once all the setup part is completed, we are ready to launch Cuckoo. We need four terminal windows, since we need different components in order to use our analysis system. There are four commands that need to be launched, in the following order:

```
sudo ./inetsim
cuckoo rooter --sudo
cuckoo
cuckoo api
```

Now, if everything worked smoothly, you have a Cuckoo instance running which is ready to receive commands from its API service. By default the API service runs on localhost on port 8090.

## A.3 Installing and configuring the CuckooDroid sandbox

Let's now move onto the Android part. This time the Virtual Machine will be a Linux one, configured as the one reported in Tab. A.3

| **Linux VM** | |
|---|---|
| Name | cuckoodroid-14-aol |
| Guest OS | Ubuntu 14.04 (64 bit) |
| Memory size | 16384 MB |
| Number of CPUs | 8 |
| Shared folders | None |
| Network type | Host-only |

Table A.2. Configuration of the Linux Virtual Machine.

Differently from the standard Cuckoo, here we can download CuckooDroid from its Github page. We need to substitute the requirements file, which is the one specifying the Python dependencies, with the one provided in our repository, named *cuckoodroidrequirements.txt*. After that, we can proceed to modify the configuration files as specified on the Cuckoodroid documentation.

### A.3.1 Configuring the Linux VM

Once you have a VM configured as the one specified in Tab. A.3, you can start adding the proper software to make it work with Cuckoo. Firstly, we need to install some Linux dependencies. These can be done through the following command:

```
sudo apt-get install libstdc++6:i386 libgcc1:i386 zlib1g:i386 libncurses5:i386
```

Now we need to add some Java dependencies. Unfortunately Ubuntu did not manage to find them by default, so we need to modify the file */etc/apt/sources.list*, adding the following lines:

```
deb http://archive.ubuntu.com/ubuntu/ trusty main restricted universe
    multiverse
deb http://archive.ubuntu.com/ubuntu/ trusty-security main restricted
    universe multiverse
deb http://archive.ubuntu.com/ubuntu/ trusty-updates main restricted universe
    multiverse
deb http://archive.ubuntu.com/ubuntu/ trusty-proposed main restricted
    universe multiverse
deb http://archive.ubuntu.com/ubuntu/ trusty-backports main restricted
    universe multiverse
```

Then, get back to a terminal and type:

```
sudo apt-get update
sudo apt-get install openjdk-7-jre
```

This will probably not work on the first attempt, it will require some additional libraries. Just copy the name of those libraries and run an *apt-get install* until you get them all.
The next step is to download Android SDK. This can be done by following the instruction on CuckooDroid's manual (we did not follow it in the previous steps since it was needed a workaround to overcome the dependencies problem). All the subsequent steps can be done by simply following the manual, starting from the previous link.

### A.3.2 Running CuckooDroid

In order to launch Cuckoodroid you just need to open a pair of terminals and launch, from the cuckoo folder, the following commands:

```
python cuckoo.py
python utils/api.py
```

## A.4 Filling customization files

It is now time to configure our analysis system to perform the kind of analysis that you want. Among the files provided in the code repository, there is a config.py which contains some configuration variables that needs to be modified before starting the analysis. Let's see what each of them mean:

- **PLATFORM**: it tells the system if we want to analyze *Windows* or *Android* malware samples.

- **MODE**: it tells if we want to extract *static* or *dynamic* features.
- **MACHINE**: name of the Cuckoo machine. In our case it could be either *cuckoo1-win7* and *cuckoodroid-14-aol*, but the user must insert the ones it used when configured Cuckoo.
- **FAMILIES**: it is a Python list of strings, containing the names of the families that will be analyzed. These names must be exactly the same as the ones used in the dataset folder.
- **D_SOURCE**: path to the dataset folder.
- **MAX_SAMPLES_DW**: this is used only for dynamic analysis on Windows samples. When we used the system on our dataset we processed pools of 100 samples and this is the default value, but you can change it through this variable.

The last thing to insert is the Cuckoo API key, which can be found inside the configuration file placed in *cuckoo/conf/cuckoo.conf* under the voice *api_token*. This must be placed inside the file *mysecrets.py*. It is needed to communicate with Cuckoo's API.

### A.4.1 Customizing Cuckoo

It is also possible to customize the sandbox environment. This can be done through Cuckoo's configuration files, which are located in the *conf* directory (under the Cuckoo main folder). You can follow Cuckoo and CuckooDroid documentation to know what is possible to modify through these files (some settings require additional tools).

## A.5 Samples format

Our analysis systems supports two kinds of files: Portable Executable (PE) and Android Package (APK). These may be also contained inside a ZIP archive. You can use our datasets, which can be requested, or your own samples. This should be contained inside a single folder, which will be specified in the *config.py*, and then there should be a folder for each family you want to analyze.

## A.6 Launching an analysis

Once all the previous steps have been completed, hence you have all the required tools properly running and the analysis system properly configured, you can launch an analysis. This can be started by just typing, in the main repository:

```
python3 analyzer.py
```

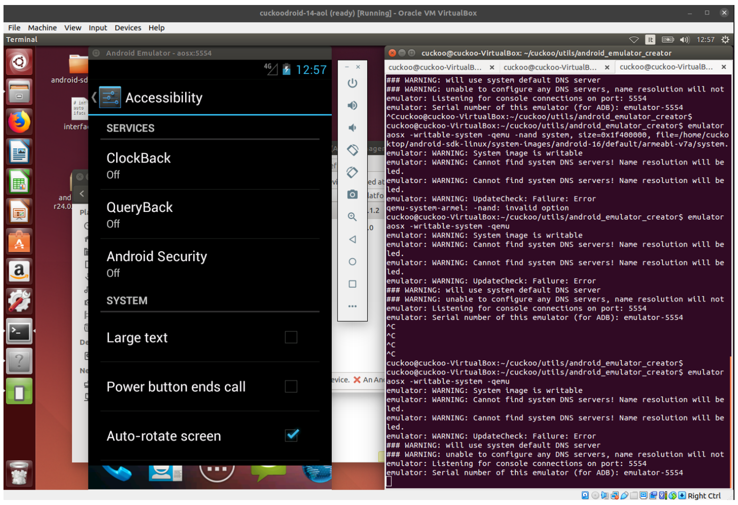
The analyzer will start grabbing samples from the provided folders, submit them to cuckoo and start waiting for results to be produced. Once an analysis is completed, its related files are removed (to avoid memory saturation) and only the relevant parts (i.e. features) will be saved in proper files. These files are located inside the *results* folder, where there will be one folder for each family and inside of that, one folder for each wanted feature.
The analyzer will keep submitting samples to Cuckoo until they have been depleted or the desired amount of results have been obtained. This is done because some samples may not work properly, being rejected by Cuckoo, or simply they will not provide any useful result. By default, the analyzer uses all the samples found in the family's folder, but we can specify one if we want to analyze only a subset of our samples.
If, for any reason, you need to stop the analyzer you can send a SIGINT (through the kill command, or simply a CTRL+C) and some log data will be written in a dump.out file, so it is possible to check how many samples have been processed and restart from where you break (rewriting the configuration file).

### A.6.1 Run example

Let's say we want to run dynamic analysis on some Android samples which belong to two different families such as Anubis and HiddenAd. We will set the config.py file as follows:

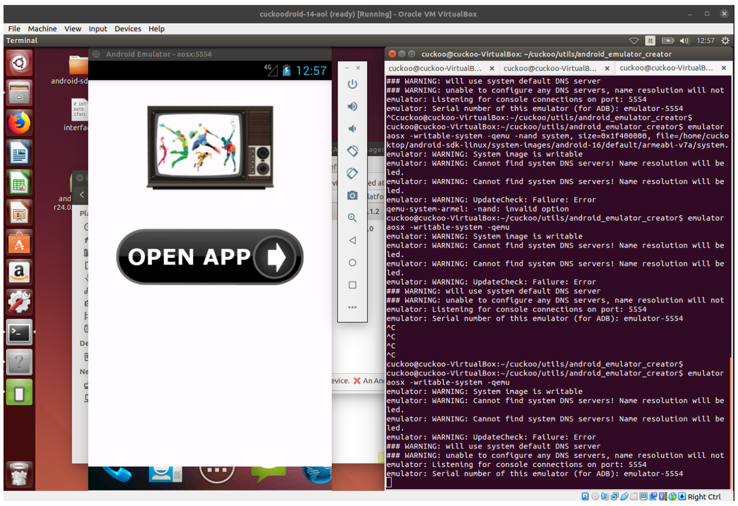
```
PLATFORM = "ANDROID"
MODE = "DYNAMIC"
MACHINE = "cuckoodroid-14-aol"
FAMILIES = ["Anubis", "HiddenAd"]
D_SOURCE = "/home/andrea/Desktop/dataset/android/"
MAX_SAMPLES_DW = -1
```

Then we start CuckooDroid, running the two commands mentioned in Sec. A.3.2. Now we just have to open a terminal and launch, using Python, the analyzer.py script. This will create a *result* folder with all the extracted features. Then the submission phase will start and every application found in the *D_SOURCE* folder will be subjected to analysis. Now cuckoo will start spawning and shutting down the *cuckoodroid-14-aol* VM, until every sample has been processed. Figs. A.1 and A.2 show two VMs while they are executing a sample.


cuckoodroid-14-aol (ready) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
Android Emulator - aosx:5554
12:57
Accessibility
SERVICES
ClockBack
Off
QueryBack
Off
Android Security
Off
SYSTEM
Large text
Power button ends call
Auto-rotate screen
cuckoo@cuckoo-VirtualBox: ~/cuckoo/utils/android_emulator_creator
Right Ctrl

```
### WARNING: will use system default DNS server
### WARNING: unable to configure any DNS servers, name resolution will not
emulator: Listening for console connections on port: 5554
emulator: Serial number of this emulator (for ADB): emulator-5554
^Ccuckoo@cuckoo-VirtualBox:~/cuckoo/utils/android_emulator_creator$
cuckoo@cuckoo-VirtualBox:~/cuckoo/utils/android_emulator_creator$ emulator
aosx -writable-system -qemu -nand system, size=0x1f400000, file=/home/cucko
ktop/android-sdk-linux/system-images/android-16/default/armeabi-v7a/system.
emulator: WARNING: System image is writable
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: UpdateCheck: Failure: Error
qemu-system-armel: -nand: invalid option
cuckoo@cuckoo-VirtualBox:~/cuckoo/utils/android_emulator_creator$ emulator
aosx -writable-system -qemu
emulator: WARNING: System image is writable
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: UpdateCheck: Failure: Error
### WARNING: will use system default DNS server
### WARNING: unable to configure any DNS servers, name resolution will not
emulator: Listening for console connections on port: 5554
emulator: Serial number of this emulator (for ADB): emulator-5554
^C
^C
^C
^C
cuckoo@cuckoo-VirtualBox:~/cuckoo/utils/android_emulator_creator$
cuckoo@cuckoo-VirtualBox:~/cuckoo/utils/android_emulator_creator$ emulator
aosx -writable-system -qemu
emulator: WARNING: System image is writable
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: UpdateCheck: Failure: Error
### WARNING: will use system default DNS server
### WARNING: unable to configure any DNS servers, name resolution will not
emulator: Listening for console connections on port: 5554
emulator: Serial number of this emulator (for ADB): emulator-5554
```

Figure A.1. Linux VM executing an Anubis sample.


cuckoodroid-14-aol (ready) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
Android Emulator - aosx:5554
12:57
OPEN APP
cuckoo@cuckoo-VirtualBox: ~/cuckoo/utils/android_emulator_creator
Right Ctrl

```
### WARNING: will use system default DNS server
### WARNING: unable to configure any DNS servers, name resolution will not
emulator: Listening for console connections on port: 5554
emulator: Serial number of this emulator (for ADB): emulator-5554
^Ccuckoo@cuckoo-VirtualBox:~/cuckoo/utils/android_emulator_creator$
cuckoo@cuckoo-VirtualBox:~/cuckoo/utils/android_emulator_creator$ emulator
aosx -writable-system -qemu -nand system, size=0x1f400000, file=/home/cucko
ktop/android-sdk-linux/system-images/android-16/default/armeabi-v7a/system.
emulator: WARNING: System image is writable
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: UpdateCheck: Failure: Error
qemu-system-armel: -nand: invalid option
cuckoo@cuckoo-VirtualBox:~/cuckoo/utils/android_emulator_creator$ emulator
aosx -writable-system -qemu
emulator: WARNING: System image is writable
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: UpdateCheck: Failure: Error
### WARNING: will use system default DNS server
### WARNING: unable to configure any DNS servers, name resolution will not
emulator: Listening for console connections on port: 5554
emulator: Serial number of this emulator (for ADB): emulator-5554
^C
^C
^C
^C
cuckoo@cuckoo-VirtualBox:~/cuckoo/utils/android_emulator_creator$
cuckoo@cuckoo-VirtualBox:~/cuckoo/utils/android_emulator_creator$ emulator
aosx -writable-system -qemu
emulator: WARNING: System image is writable
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: Cannot find system DNS servers! Name resolution will be
led.
emulator: WARNING: UpdateCheck: Failure: Error
### WARNING: will use system default DNS server
### WARNING: unable to configure any DNS servers, name resolution will not
emulator: Listening for console connections on port: 5554
emulator: Serial number of this emulator (for ADB): emulator-5554
```

Figure A.2. Linux VM executing an HiddenAd sample.

# Appendix B

# Developer manual

The code repository consists of various files, which have been briefly described in the user manual. Let's take a deeper look to those files that contains actual code:

- analyzer.py: this mainly contains functions that are in charge of grabbing samples from the dataset directory and start the analysis.
- clientAPI.py: it implements some methods to contact Cuckoo API's endpoints.
- android_static.py: this file contains the code to interact with Droidlysis (static analysis of android samples).
- extractFeatures.py: it contains the code used by the static analyzer as well as some utility methods, which can be used to extract features from reports that Cuckoo generates.
- conf.py: it contains configuration variables which have been described in the user manual.
- mysecrets.py: here it is placed the Cuckoo's API key.

Let's now see the methods implemented in each of these files.

**analyzer.py**

- log(msg):
  writes a specific message to the logfile.
  :param msg: message to be written.
- getAvailable(tag):
  returns the number of samples available for a specific family.
  :param tag: family's name.
- waitForReport(taskId):
  puts the system to sleep while waiting for a report to be produced.
  :param taskId: id of the analysis' task.
- parseReportDynamic(taskId):
  read API's counts from a report.
  :param taskId: id of the analysis' task.
- saveResults(res, tag, sample, type):
  saves features on the *results* directory.
  :param res: feature's dictionary.
  :param tag: family's name.
  :param sample: sample's name.
  :param type: selected feature.

- sendToCuckoo(source, tag, analyses, samples, limit=50, start=0, ext=""):
  takes a sample from the dataset directory and sends it to Cuckoo.
  :param source: path to sample.
  :param tag: family's name.
  :param analyses: list of completed analyses.
  :param samples: list of analyzed samples.
  :param limit: limit of samples to be taken from the dataset folder.
  :param start: index of the dataset folder from which to start grabbing samples.
  :param ext: file extension.

- analyzeWithCuckoo(tag, limit, start, analyses, samples):
  iterates the previous function until every sample has been processed.
  :param tag: family's name.
  :param limit: limit of samples to be taken from the dataset folder.
  :param start: index of the dataset folder from which to start grabbing samples.
  :param analyses: list of completed analyses.
  :param samples: list of analyzed samples.

- analyzeWithCuckooDroid(tag, analyses, samples):
  iterates the sendToCuckoo function on each sample of the Android collection.
  :param tag: family's name.
  :param analyses: list of completed analyses.
  :param samples: list of analyzed samples.

- analyzeStatic(tag):
  extract static features from a family.
  :param tag: family's name.

- getCollected(tag):
  returns a list of all the samples analyzed (their file name).
  :param tag: family's name.

- analyzeW():
  wrapper for the analyzeWithCuckoo function.

- analyzeA():
  wrapper for the analyzeWithCuckooDroid function.

- prepareDyn():
  creates the result folder.

**clientAPI.py**

- submitSample(path, machineId, timeout="", package="exe"):
  contacts the endpoint *tasks/create/file* to submit a sample.
  :param path: sample's path. Returns the task identifier. :param machineId: Virtual Machine's identifier.
  :param timeout: duration of the analysis.
  :param package: sample's file format.

- getTaskState(taskId):
  returns the state of a task.
  :param taskId: task's identifier.

- getCuckooStatus():
  returns the status of Cuckoo (i.e. number of connected VM, their status etc.).

- retrieveReport(taskId):
  returns an analysis' report in JSON format.
  :param taskId: task's identifier.

- cleanData(taskId):
  cleans every data generated by Cuckoo for a specific task.
  :param taskId: task's identifier.
- shutdownServer():
  shuts down Cuckoo.

**android_static.py**

- saveResults(sample, feats, tag):
  saves extracted features in proper files under the *results* folder.
  :param sample: sample's name.
  :param feats: list of extracted features.
  :param tag: family's name.
- extractStaticFeatures(res, feat, sample, tag):
  extracts static features from the Droidlysis report.
  :param res: report (python dictionary).
  :param feat: list of extracted features.
  :param sample: sample's name.
  :param tag: family's name.
- prepareStatic():
  creates the results folder.
- analyzeStatic(sample, tag, features):
  launches Droidlysis and extracts features.
  :param sample: sample's name.
  :param tag: family's name.
  :param features: list of extracted features.
- parseCollection(tag, features):
  iterates the analysis on each sample of a specific family.
  :param tag: family's name.
  :param features: list of extracted features.
- runAndroidStatic():
  launches static analysis on each Android family.

**extractFeatures.py**

- extractArchive(source, ext=""):
  extracts samples from the dataset directory. It can handles *.zip, .7z, .apk, .exe* files. It returns the sample's name.
  :param source: sample's path.
  :param ext: extension of the file, it is used only if the sample's name does not contain its extension.
- extractStaticAPIs(source, exe=True):
  extracts APIs declared in the Import Access Table of Windows executables. Returns them into a list.
  :param source: sample's path.
  :param exe: tells if the features are going to be extracted from the original sample or from the Cuckoo report.
- extractSectionEntropy(source, exe=True):
  extracts sections' entropy from Windows executables. Returns them into a dictionary.
  :param source: sample's path.
  :param exe: tells if the features are going to be extracted from the original sample or from the Cuckoo report.

- extractDynamicAPIs(report):
  returns a dictionary with the number of calls performed for each API.
  :param report: portion of the Cuckoo report containing the APIs.
- trimHead(sample):
  removes headers from PEs and writes the remaining bytes into a file. Returns the name of that file.
  :param sample: sample's path.
- prepareStaticW():
  creates the folder for static features.

# Bibliography

[1] SonicWall, "Sonicwall cyber threat report 2023.", https://www.sonicwall.com/medialibrary/en/white-paper/2023-cyber-threat-report.pdf

[2] Ghidra, https://ghidra-sre.org/

[3] REMnux: A Linux Toolkit for Malware Analysis, https://remnux.org/

[4] Microsoft Sysinternal Suite, https://learn.microsoft.com/en-us/sysinternals/downloads/sysinternals-suite

[5] E. Amer and I. Zelinka, "A dynamic windows malware detection and prediction method based on contextual understanding of api call sequence", Computers & Security, vol. 92, 2020, p. 101760, DOI https://doi.org/10.1016/j.cose.2020.101760

[6] D. Serpanos, P. Michalopoulos, G. Xenos, and V. Ieronymakis, "Sisyfos: A modular and extendable open malware analysis platform", Applied Sciences, vol. 11, no. 7, 2021, p. 2980

[7] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences", AI 2016: Advances in Artificial Intelligence: 29th Australasian Joint Conference, Hobart, TAS, Australia, December 5-8, 2016, Proceedings 29, 2016, pp. 137–149

[8] NIST, "Malware definition", https://csrc.nist.gov/glossary/term/malware

[9] Kaspersky, "Malware definition", https://www.kaspersky.com/resource-center/preemptive-safety/what-is-malware-and-how-to-protect-against-it

[10] CISCO, "Malware definition", https://www.cisco.com/c/en/us/products/security/advanced-malware-protection/what-is-malware.html

[11] "What is malware? the ultimate guide to malware", https://www.avg.com/en/signal/what-is-malware

[12] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern eraa state of the art survey", ACM Computing Surveys (CSUR), vol. 52, no. 5, 2019, pp. 1–48

[13] Kaspersky, "Ransomware types", https://www.kaspersky.it/resource-center/threats/ransomware-attacks-and-types

[14] ENISA, "Rootkit types", https://www.enisa.europa.eu/topics/csirts-in-europe/glossary/rootkits

[15] "Lockbit 3.0 update — unpicking the ransomwares latest anti-analysis and evasion techniques", https://www.sentinelone.com/labs/lockbit-3-0-update-unpicking-the-ransomwares-latest-anti-analysis-and-evasion-techniques/

[16] "Eternalblue", https://www.cisecurity.org/wp-content/uploads/2019/01/Security-Primer-EternalBlue.pdf

[17] "Windows privilege escalation: Spoolfool", https://www.hackingarticles.in/windows-privilege-escalation-spoolfool/

[18] "Threat analysis report: Lockbit 2.0 - all paths lead to ransom", https://www.cybereason.com/blog/threat-analysis-report-lockbit-2.0-all-paths-lead-to-ransom

[19] T. Lair, "Exploiting environment variables in scheduled tasks for uac bypass", https://www.tiraniddo.dev/2017/05/exploiting-environment-variables-in.html

[20] "Explained: Domain generating algorithm", https://blog.malwarebytes.com/security-world/2016/12/explained-domain-generating-algorithm/

[21] J. Singh and J. Singh, "Challenge of malware analysis: malware obfuscation techniques", International Journal of Information Security Science, vol. 7, no. 3, 2018, pp. 100–110

[22] R. Tahir, "A study on malware and malware detection techniques", International Journal of Education and Management Engineering, vol. 8, no. 2, 2018, p. 20

[23] R. R. Branco, G. N. Barbosa, and P. D. Neto, "Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies", Black Hat, vol. 1, no. 2012, 2012, pp. 1–27

[24] A. Sharma, B. B. Gupta, A. K. Singh, and V. Saraswat, "Orchestration of apt malware evasive manoeuvers employed for eluding anti-virus and sandbox defense", Computers & Security, vol. 115, 2022, p. 102627, DOI https://doi.org/10.1016/j.cose.2022.102627

[25] 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

[26] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern eraa state of the art survey", ACM Comput. Surv., vol. 52, sep 2019, DOI 10.1145/3329786

[27] M. Carpenter, T. Liston, and E. Skoudis, "Hiding virtualization from attackers and malware", IEEE Security & Privacy, vol. 5, no. 3, 2007, pp. 62–65, DOI 10.1109/MSP.2007.63

[28] StatCounter, "Desktop operating system market share worldwide", https://gs.statcounter.com/os-market-share/desktop/worldwide

[29] Statista, "Operating systems most affected by malware as of 1st quarter 2020", https://www.statista.com/statistics/680943/malware-os-distribution/

[30] StatCounter, "Mobile operating system market share worldwide", https://gs.statcounter.com/os-market-share/mobile/worldwide

[31] Kaspersky, "Mobile security: Android vs ios which one is safer?", https://www.kaspersky.com/resource-center/threats/android-vs-iphone-mobile-security

[32] M. Sikorski and A. Honig, "Practical malware analysis: the hands-on guide to dissecting malicious software", no starch press, 2012

[33] M. S. Kirmani and M. T. Banday, "Analyzing detection avoidance of malware by process hiding", 2018 3rd International Conference on Contemporary Computing and Informatics (IC3I), 2018, pp. 293–297

[34] Android, "Android list of permissions", https://developer.android.com/guide/topics/permissions/overview

[35] Y. Fratantonio, C. Qian, S. Chung, and W. Lee, "Cloak and Dagger: From Two Permissions to Complete Control of the UI Feedback Loop", Proceedings of the IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, May 2017

[36] "Cloak & dagger", https://cloak-and-dagger.org/

[37] R. Harang and E. M. Rudd, "Sorel-20m: A large scale benchmark dataset for malicious pe detection", 2020

[38] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge", arXiv preprint arXiv:1802.10135, 2018

[39] Malware Bazaar, https://bazaar.abuse.ch/about/

[40] Virus Share, https://virusshare.com/

[41] VX-underground, https://www.vx-underground.org/

[42] Malpedia, https://malpedia.caad.fkie.fraunhofer.de/

[43] O. Or-Meir, N. Nissim, Y. Elovici, and L. Rokach, "Dynamic malware analysis in the modern eraa state of the art survey", ACM Computing Surveys (CSUR), vol. 52, no. 5, 2019, pp. 1–48

[44] Microsoft, https://learn.microsoft.com/en-us/windows/security/threat-protection/windows-sandbox/windows-sandbox-overview

[45] S. Agarwal and G. Raj, "Frame: framework for real time analysis of malware", 2018 8th International Conference on Cloud Computing, Data Science & Engineering (Confluence), 2018, pp. 14–15

[46] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools", ACM computing surveys (CSUR), vol. 44, no. 2, 2008, pp. 1–42

[47] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and lstm", Multimedia Tools and Applications, vol. 78, no. 4, 2019, pp. 3979–3999

[48] S. Shakya and M. Dave, “Analysis, detection, and classification of android malware using system calls”, arXiv preprint arXiv:2208.06130, 2022

[49] S. Gupta, H. Sharma, and S. Kaur, “Malware characterization using windows api call sequences”, Security, Privacy, and Applied Cryptography Engineering (C. Carlet, M. A. Hasan, and V. Saraswat, eds.), Cham, 2016, pp. 271–280

[50] ApiMonitor, https://www.apimonitor.com/

[51] Y. B. Lee, J. H. Suk, and D. H. Lee, “Bypassing anti-analysis of commercial protector methods using dbi tools”, IEEE Access, vol. 9, 2021, pp. 7655–7673

[52] Intel Pin - A Dynamic Binary Instrumentation Tool, https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[53] DynamoRio, https://dynamorio.org/

[54] M. Polino, A. Continella, S. Mariani, S. DAlessio, L. Fontata, F. Gritti, and S. Zanero, “Measuring and Defeating Anti-Instrumentation-Equipped Malware”, Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2017

[55] x64dbg, “Windows debugger”, https://x64dbg.com/

[56] Microsoft Windbg, https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools

[57] ApkTool, https://ibotpeaches.github.io/Apktool/

[58] Android Studio, https://developer.android.com/studio

[59] DrlTrace, https://github.com/mxmssh/drltrace

[60] D. C. D’Elia, S. Nicchi, M. Mariani, M. Marini, and F. Palmaro, “Designing robust api monitoring solutions”, IEEE Transactions on Dependable and Secure Computing, no. 01, 2021, pp. 1–1

[61] P. F. Tirenna, “Techniques for malware analysis based on symbolic execution”, 2020

[62] Hex Rays, IDA Pro, https://hex-rays.com/ida-pro/

[63] D. Plohmann and A. Hanel, “simplifire. idascope”, 2012

[64] Mandiant, Flare IDA, https://github.com/mandiant/flare-ida

[65] dex2jar, https://github.com/pxb1988/dex2jar

[66] Process Monitor, https://learn.microsoft.com/en-us/sysinternals/downloads/procmon

[67] Android Debug Bridge, https://developer.android.com/studio/command-line/adb

[68] MonkeyRunner, https://developer.android.com/studio/test/monkeyrunner

[69] Cuckoo Sandbox, https://cuckoosandbox.org/

[70] S. Jamalpur, Y. S. Navya, P. Raja, G. Tagore, and G. R. K. Rao, “Dynamic malware analysis using cuckoo sandbox”, 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), 2018, pp. 1056–1060, DOI 10.1109/ICICCT.2018.8473346

[71] M. Ijaz, M. H. Durad, and M. Ismail, “Static and dynamic malware analysis using machine learning”, 2019 16th International Bhurban Conference on Applied Sciences and Technology (IBCAST), 2019, pp. 687–691, DOI 10.1109/IBCAST.2019.8667136

[72] Cuckoo Droid Sandbox, https://cuckoo-droid.readthedocs.io/en/latest/#

[73] AnyRun Sandbox, https://any.run/

[74] Triage Sandbox, https://tria.ge/

[75] Statista, “Mobile operating systems’ market share worldwide from 1st quarter 2009 to 4th quarter 2022”, https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/

[76] Statista, “Global market share held by operating systems for desktop pcs, from january 2013 to june 2022”, https://www.statista.com/statistics/218089/global-market-share-of-windows-7/

[77] Kaspersky, “Mobile malware evolution 2021”, https://securelist.com/mobile-malware-evolution-2021/105876/

[78] MalwareBytes, “2022 threat review”, https://www.malwarebytes.com/resources/malwarebytes-threat-review-2022/mwb_threatreview_2022_ss_v1.pdf

[79] Sophos, “The state of ransomware 2022”, https://assets.sophos.com/X24WTUEQ/at/4zpw59pnkpxxnhfhgj9bxgj9/sophos-state-of-ransomware-2022-wp.pdf

[80] Caro, "A new virus naming convention", http://www.caro.org/articles/naming.html

[81] L. Wang, H. Wang, R. He, R. Tao, G. Meng, X. Luo, and X. Liu, "Malradar: Demystifying android malware in the new era", Proceedings of the ACM on Measurement and Analysis of Computing Systems, vol. 6, no. 2, 2022, pp. 1–27

[82] VX-underground, "Argus apk collection", https://samples.vx-underground.org/samples/Blocks/Argus%20Collection/

[83] A. H. Lashkari, A. F. A. Kadir, L. Taheri, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark android malware datasets and classification", 2018 International Carnahan Conference on Security Technology (ICCST), 2018, pp. 1–7, DOI 10.1109/CCST.2018.8585560

[84] C. Wei, Q. Li, D. Guo, and X. Meng, "Toward identifying apt malware through api system calls", Security and Communication Networks, vol. 2021, 2021, pp. 1–14

[85] VX-underground, "In the wild collection", https://samples.vx-underground.org/samples/Blocks/InTheWild%20Collection/

[86] M. S. I. Microsoft, "Win32/zbot description", https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/Zbot&threatId=

[87] A. Ionescu, "Closing heavens gate.", https://www.alex-ionescu.com/?p=300

[88] M. S. I. Microsoft, "Win32/trickbot description", https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Win32/Trickbot&threatId=

[89] Kaspersky, "Trickster description", https://threats.kaspersky.com/en/threat/Trojan.Win32.Trickster/

[90] M. S. I. Microsoft, "Win32/ramnit description", https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Trojan%3aWin32%2fRamnit.D

[91] C. P. t. Micha Praszmo, "Ramnit- in-depth analysis.", https://www.cert.pl/en/posts/2017/09/ramnit-in-depth-analysis/

[92] "Mobile malware analysis : Tricks used in anubis", https://eybisi.run/Mobile-Malware-Analysis-Tricks-used-in-Anubis/

[93] K. Roman Unuchek, "A new era in mobile banking trojans", https://securelist.com/a-new-era-in-mobile-banking-trojans/79198/

[94] K. Roman Unuchek, "The asacub trojan: from spyware to banking malware.", https://securelist.com/the-asacub-trojan-from-spyware-to-banking-malware/73211/

[95] K. Tatyana Shishkova, "The rise of mobile banker asacub", https://securelist.com/the-rise-of-mobile-banker-asacub/87591/

[96] K. Victor Chebyshev, "Hqwar: the higher it flies, the harder it drops", https://securelist.com/hqwar-the-higher-it-flies-the-harder-it-drops/93689/

[97] A. Consulting, "The stop ransomware variant", https://angle.ankura.com/post/102het9/the-stop-ransomware-variant

[98] D. Lama, "Malware ransomware report - wannacry ransomware", https://github.com/0xZuk0/rules-of-yaras/blob/main/reports/Wannacry%20Ransomware%20Report.pdf

[99] "Droidlysis, a property extractor for android apps.", https://github.com/cryptax/droidlysis

[100] Ero Carrera, "Pefile, a multi-platform python module to parse and work with portable executable (pe) file", https://github.com/erocarrera/pefile

[101] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges", Journal of Network and Computer Applications, vol. 153, 2020, p. 102526

[102] A. Abusitta, M. Q. Li, and B. C. Fung, "Malware classification and composition analysis: A survey of recent developments", Journal of Information Security and Applications, vol. 59, 2021, p. 102828

[103] R. B. Hadiprakoso, H. Kabetta, and I. K. S. Buana, "Hybrid-based malware analysis for effective and efficiency android malware detection", 2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS), 2020, pp. 8–12

[104] Android, "Android developers documentation, manifest.permissions", https://developer.android.com/reference/android/Manifest.permission