POLITECNICO
MILANO 1863

Facoltà di Ingegneria Industriale e dell'Informazione
Master's Degree in Computer Science and Engineering

An evaluation of anti-evasion techniques implemented in malware analysis sandboxes and debuggers

Master of Science thesis of:
Salvatore Bova matricola 912868

Advisor:
Prof. Stefano Zanero.

Co-advisor:
Michele Carminati, PhD.
Mario Polino, PhD.

Salvatore Bova: An evaluation of anti-evasion techniques implemented in malware analysis sandboxes and debuggers

ACKNOWLEDGEMENTS

I would like to thank:

My advisor, prof. Stefano Zanero.
My co-advisors Michele Carminati and Mario Polino.
All the people I met during my university career.

For their support and guidance throughout my work.

All my family members, especially my parents that have always supported and encouraged me, and also my second family composed of all my friends, this whole journey would not have been possible without you.

ABSTRACT

Malware is still today a major worldwide problem. In the last years, profit became the target of malware authors and the constant digitalization growth of many companies led to a new source of gain for cybercriminals. To protect simple users and companies from cyber threats it is necessary to know and to understand them. This led to the born of malware analysis that is one of the most important methods against this threat. Due to the large number of malware samples, doing the classical steps of malware analysis manually is time-consuming for malware analysts, therefore they developed many automated malware analysis systems. Most of these systems consist of a sandbox that runs the malware in an isolated environment and provides static and behavioral information, but the battle with malware never ends, while new technologies and processes evolve rapidly to deal with it, so too do those of the attackers. Indeed these sandbox environments often leave specific artifacts which can be used by malware to determine if it is being executed in a virtual environment, therefore malware authors implement different techniques, the so-called *evasion techniques* that are used to evade systems analysis by the malware. On the other side, to fight back, sandbox authors try to implement different methods to detect if malware is trying to do evasion and to nullify the evasion-techniques, these methods can be called *anti-evasion techniques*.
In this thesis, we study how the security specialists try to defeat the evasion techniques, analyzing the behavior of the most important public sandboxes available on the Internet. We also collect information about debuggers, which are important tools used during dynamic malware analysis, and how they try to defeat evasion techniques. To obtain this result, we collected, implemented, and tested more than 130 evasion-techniques. To test this thesis, we implemented a tool that allows us to apply these techniques to a malware sample at choice, in this way we can analyze directly the effectiveness of each technique against the sandboxes. Based on the results of our experiments we show the current strength of the automated malware analysis systems. We also show how certain evasion-techniques families are more useful than others against an automated analysis system, and we demonstrate how strong evasion-techniques against automated analysis systems could be instead easily detected by a human analyst.

SOMMARIO

I malware sono tutt'oggi uno dei più grandi problemi nel mondo dell'informatica, ma considerando l'attuale avanzamento tecnologico, il rapido sviluppo e il ruolo fondamentale che l'informatica sta assumendo attraverso il suo utilizzo in quasi tutti gli aspetti della nostra società e i sempre più grandi capitali che si stanno investendo nella digitalizzazione i malware non sono più solo un problema relegato al mondo IT, ma possono causare gravi danni all'economia, alla sanità e a molti altri servizi pubblici. Gli autori di malware negli ultimi anni hanno fatto del profitto il loro maggior obiettivo, organizzandosi in veri e propri gruppi criminali che sfruttano le loro conoscenze informatiche per generare profitto recando danni a banche, istituzioni o aziende private con l'utilizzo, molto spesso, di software malevoli. Questo nuovo trend ha portato alla necessità di dover analizzare e studiare i malware in modo da poter difendersi, buona parte dell'analisi dei malware però si basa sul reverse engineering manuale o sull'analisi statica, usando software come gli antivirus in grado di rilevare, tramite "signature", l'eventuale presenza di codice malevolo. Tuttavia, a causa dell'elevato numero di malware generato ogni giorno questi approcci non sono efficaci o comunque richiedono tempi eccessivamente lunghi per andare a buon termine. Per questo gli specialisti di sicurezza informatica hanno sviluppato dei tool automatici per l'analisi dei file sospetti, molti di questi tool usano una sandbox per eseguire il file in sicurezza e alcuni di loro sono gratuitamente accessibili dal web e forniscono un report basato sull'analisi comportamentale. Ma all'interno delle sandbox ci sono degli artefatti, ovvero dei file essenziali per il funzionamento della sandbox stessa, che possono essere utilizzati dai malware per determinare se sono sotto analisi o meno. Quindi per evitare quest'analisi, gli autori di malware hanno inziato a sviluppare malware evasivi, ovvero dei malware che possono capire, attraverso diverse tecniche conosciute come *tecniche di evasione*, se sono sotto analisi ed in quel caso nascondere la loro natura malevola. Con il lavoro sviluppato in questa tesi vogliamo valutare la resistenza dei più importanti tool per l'analisi automatica di malware gratuitamente accessibili dal web. Uno dei nostri principali obiettivi è di capire in che modo e con quali tecniche gli autori di questi tool cercano di contrastare le tecniche di evasione. Per fare ciò abbiamo raccolto e implementato più di 130 diverse tecniche di evasione e abbiamo sviluppato un tool che ci permette di applicarle ad un file qualsiasi in modo da poter testarne

direttamente l'efficacia contro le analisi effettuate in sandbox. Con i risultati ottenuti dalle varie sandbox disponibili online, e dai più importanti debuggers per Windows, che abbiamo deciso di analizzare in quanto strumenti fondametali durante l'analisi dinamica dei malware, abbiamo anche classificato in base alla loro efficacia le varie tecniche di evasione.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

API Application programming interface

VM Virtual Machine

WMI Windows Management Instrumentation

WQL Windows Management Instrumentation Query Language

PEB Process Environment Block

OS Operating System

1 INTRODUCTION

The first malware, intended as software intentionally designed to cause damage to a computer system, in computer history was written in the 70s. In the first years of malware history, they were developed usually just to show off the author's skills or to demonstrate that practical implementations of theoretic notions, like self-reproducing code or self-obfuscation, were possible. In the beginning, they spread on personal computers by infecting executable programs or boot sectors of floppy disks, but by the mid-90s, however, with the spread of the Internet, the malware began to spread much faster, using the Internet and e-mail exchange as a source for new infections.
This change led to new aspects in malware development, the new target of malware developers was mass attacks also known as opportunistic attacks, which involve malware that are distributed in large numbers for anyone to download or to be injected into websites easy for anyone to access.

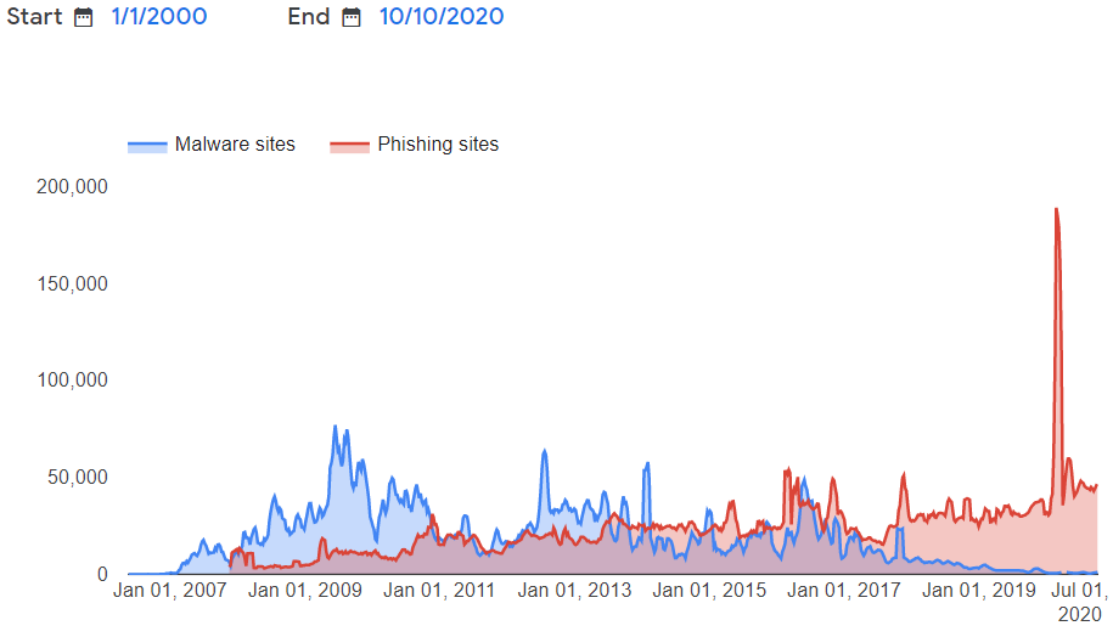


Figure 1: Unsafe websites detected per week

How we can see from Figure 1, obtained from Google Datasets [11], from 2009 malware spread all over the Internet, this so-called mass-malware were written by organized groups to capture and transmit users' private or sensitive information, so they were written to infect the largest possible number of users to increase the gain of the authors.
So malware now are profit-oriented, but in the last years against normal users usually are used pishing websites which pretend to be legitimate so that they can trick users into typing in their usernames and passwords or sharing other private information. While cybercriminals use malware with the aim of strategical attacks against critical infrastructures or high-profile targets, for example, the 64% of companies have experienced web-based attacks and the average cost of a malware attack for a company is $2.6 million [30]. Malware nowadays can also be used for political activism and espionage activity, this makes states new actors in malware development.
With the following graph we can understand better the damage caused by cybercrimes, usually accomplished through the use of malware.

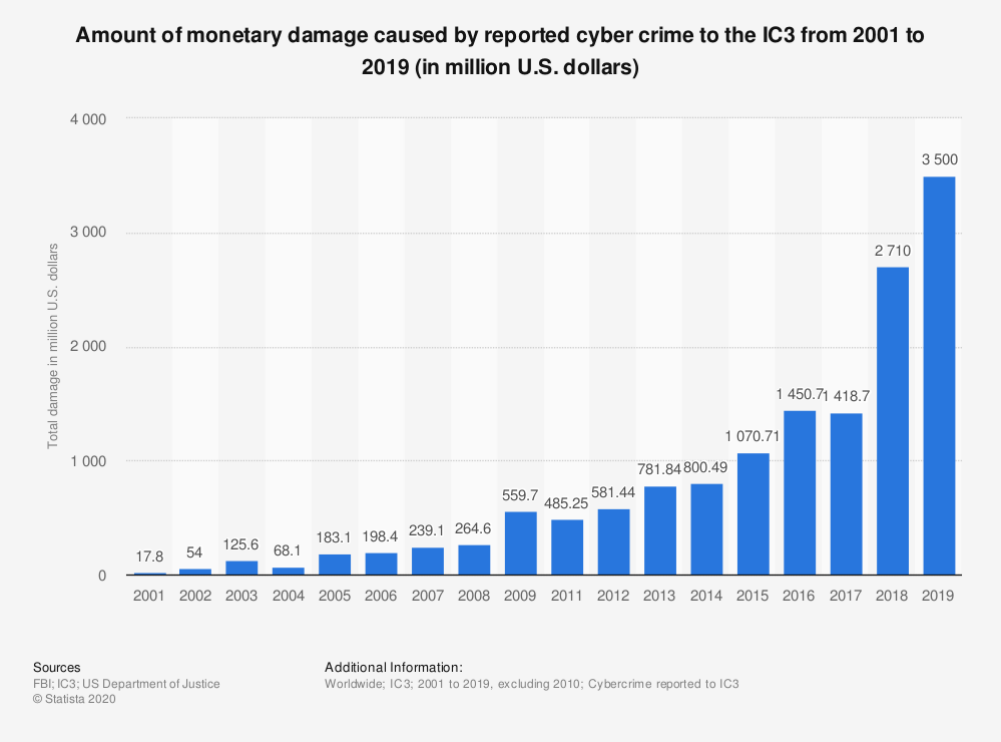


Figure 2: Total damage caused by reported cyber crime 2001-2019

This trend keeps growing, indeed the damage related to cybercrime is projected to hit $6 trillion annually by 2021 [6]. Due to this huge proliferation of malware and to the high impact of them on the economy researchers revolved their attention to malware analysis, the study of the functionality, origin, and potential impact of malware. The necessity of defense against malware led also to new software, like antivirus and anti-malware which have as basic strategy a signature-based detection. To avoid this signature-based detection, malware authors began writing polymorphic malware, usually, viruses that change the layout with each infection, so malware analysts come up with dynamic malware analysis that is performed by observing the behavior of the malware while it is actually running on a host system, usually, a sandbox environment, or debugging them while running using a debugger.
However dynamic malware analysis is a time-consuming operation for security specialists, and the presence of 350,000 new malware samples every day [12] and over 18 million websites [28] infected with malware makes unfeasible to check them all manually. So, to increase the number of malware samples that can be analyzed, cybersecurity specialists developed automated malware analysis services, some of them are freely available on the Internet: it is possible to upload any suspicious file, then this file will be executed and analyzed, in a sandbox environment, producing a report with all the information regarding the execution. However, malware authors try to avoid the analysis developing malware, that try to evade the system looking for several technical artifacts that must exist in virtualized environments to work properly. Evasive malware looks for these artifacts through different techniques and if they found them they do not act maliciously, avoiding in this way the analysis.
Our goal is to evaluate what kind of countermeasures the developers of automated malware analysis tools are using in their products against evasion techniques. We collected results about all the public sandboxes available on the Internet and the most important Windows debuggers. To obtain these results, we tested all the malware analysis tools against known evasion techniques. Indeed, we collected and implemented more than 130 evasion techniques, we developed a system that allows us to apply them to any file. Our system allows us to execute all the evasion techniques applied before the execution of the original malware sample, so the malware can be analyzed only if the evasion did not work. Hence if the uploaded file was recognized as malware we concluded that the sandbox has an anti-evasion technique for the evasion technique we applied to the file or the evasion-technique does not work anymore, otherwise, the technique is useful to do evasion.

This thesis is organized as follows:

- In Chapter 2, we introduce the evasive malware problem, we provide a deeper explanation of malware analysis and of all the tools and technique used by security analysts, focusing especially on dynamic program analysis, we talk about the state of the art of sandboxes and of our project goals.
- In Chapter 3, we first give an overview of our approach. Then, we explain in detail all the evasion technique families and the implementation details of our work, focusing above all on how the crafter works.
- In Chapter 4, we present everything concerning the analysis process: we explain the automated malware analysis tools used, and how we performed our tests, and finally, we present the results obtained with our work
- In Chapter 5, we discuss our project limitations and the related possible future works.
- In Chapter 6, we present some conclusive remarks about our project, how we developed it, and the results obtained by our tests.

2

BACKGROUND & MOTIVATION

In this chapter, we detail and formalize the focus of our thesis. In particular, we provide more information about evasive malware and how malware authors develop them, focusing on all the different ways and techniques used to evade analysis. Then we detail some of the current state-of-the-art approaches in malware analysis, showing all the different techniques and tools used by malware analysts. We also describe in broad terms how sandboxes and debuggers work. Finally, we present the goals of our work.

2.1 THE PROBLEM OF EVASIVE MALWARE

In the malware markets can be found everything, from simple programs to crack passwords, to companies offering software for surveillance and espionage. Some of these products are highly valuable; for example, some companies advertise a $1.5 million payout to anyone willing to sell zero-day vulnerabilities [1] in Apple's iOS operating system. Therefore there is a great variety of malicious software and analyze and detect them is essential for cybersecurity specialists.
Once the approach against malware was a signature-based detection, that consists of a process where a unique identifier is established about a known threat so that the threat can be identified in the future. In the case of an antivirus, it may be a unique pattern of code that attaches to a file, or it may be as simple as the hash of a known bad file. If that specific pattern, or signature, is discovered again, the file can be flagged as being infected. But malware authors came up with obfuscation techniques like polymorphism and metamorphism that make each sample different from another, so the most effective way, nowadays, to defeat malware is to analyze it to understand what is capable to do and how, for example discovering which vulnerability in the system it is exploiting so it can be fixed.

1 A computer-software vulnerability that is unknown to those who should be interested in mitigating it.

So due to the very profitable aspect of this market, the malware authors want to avoid their products being analyzed. To achieve this goal they started developing evasive malware; they consist of malware that use several techniques to actively detect if they are under analysis. The evolution of evasive malware is alarming: the 99% of malware is used for under one minute before the sample is changed to evade security software [17] and malware authors always develop new ways to improve the evasion.

Now we go deeper into all the different tricks used by malware authors to protect their malicious software by antivirus and anti-malware tools or by the security analysts.

2.1.1 *Obfuscation*

Malware authors to bypass security solutions based on signatures, developed polymorphic and metamorphic malware:

- Polymorphism consists of changing the layout/shape of the malware with each infection and usually, the payload is encrypted making static string analysis practically impossible.
- Metamorphism consists in create different "versions" of code that look different but in the end, they do the same thing. This is achieved through techniques like instruction reorder or dead code insertion.

A more advanced technique is *packing* that hides the real code of a program through one or more layers of compression/encryption, similar to polymorphism but used in more complex malware. To understand the importance of packing nowadays just consider that 80% of new malware is packed with various packers and that 50% of new malware samples are simply repacked versions of existing ones [2].To see the original code of packed malware is necessary the use of a specific unpacking routine. Without knowing it the only way to analyze the malware behavior is by executing it because at run-time an unpacking routine restores in memory and then executes the original code. To defeat this technique so is useful analyze the malware sample at runtime, to do it usually security specialists do not run the sample natively on a machine, instead, they use debuggers, or sandbox environments, but malware authors, to avoid the analysis, developed anti-virtualization and anti-debugging tricks that are able to detect if their malware is being analyzed.

2.1.2 *Evasion Techniques*

The main concept of evasion techniques is to recognize if the malware is being executed in a controlled environment and subsequently hide or alter its malicious behavior.

```
If (amIUnderAnalysis())
{
   die();
}
else
{
  beMalicious();
}
```

Figure 3: General scheme for evasion techniques.

The detection of the analysis environments can be performed in different ways: malware samples look for specific artifacts left by the analysis components (or agents). If any of such artifact is identified, malicious activity is not executed (or executed differently), how we can see from Figure 3 where the function *amIUnderAnalysis()* looks for some artifacts and if it found them simply stops the execution. Malware authors used so many different evasion-techniques that look for artifacts during the years that collecting all of them is almost impossible, for example, to avoid detection can be used also wear-and-tear artifacts as checking the recycle bin size, the number of browsers installed or also the number of cookies and bookmarks present on the environment [23] because in a sandbox it is more likely to have small values than in a system used by a normal user. But, in general, we can distinguish two big families of evasion techniques:

- Anti-debug: that are techniques that aim to find artifacts to detect if the malware is under debugging;
- Anti-VM: that are techniques that try to detect if the malware is executed inside a virtualized environment.

We focused above all on them, and we collected or implemented 135 evasion techniques dividing them based on where or how they try to get the information to understand if the malware is under analysis, obtaining so 18 families. Now, to understand better how evasion techniques work, let's take a look at each family and at their characteristics:

1. **Memory Fingerprinting**: this family contains all the techniques that check inside the memory regions of the running process to detect debuggers. These techniques work because a debugger usually leaves some traces in the memory of the debugged process modifying several system variables associated with the process under debug. For example, in the Process Environment Block (PEB), which is a user-mode data structure present in each Windows process, we can find variables designed to indicate the debugger presence.

```
1  struct _PEB {
2      0x000 BYTE InheritedAddressSpace;
3      0x001 BYTE ReadImageFileExecOptions;
4      0x002 BYTE BeingDebugged;
5      0x003 BYTE SpareBool;
6      ...
7      0x1ec void* AppCompatInfo;
8      0x1f0 _UNICODE_STRING CSDVersion;
9      0x1f8 void* ActivationContextData;
10     0x1fc void* ProcessAssemblyStorageMap;
11     0x200 void* SystemDefaultActivationContextData;
12     0x204 void* SystemAssemblyStorageMap;
13     0x208 DWORD MinimumStackCommit;
14 );
```

Listing 1: PEB structure

   In the above listing, for example, we can see how the third BYTE (line 4) of the PEB structure provides information about if the process is being debugged or not.

2. **Exception Handling**: in this family, we can find all the techniques that abuse the behavior of the program when exceptions or errors occur; indeed malware can detect the presence of a debugger setting a custom handler for certain exceptions, then it throws the correct exception, and if the routine is never executed, it means that the debugger caught the

exception.

3. **CPU Fingerprinting**: this family contains all the Anti-VM techniques that use the different features between Virtualized CPU/real CPU used with virtualization extension and CPU without virtualization to detect if the malware is executed inside a virtualized environment. This behavior can be intended, or it could be a side-effect due to incorrect execution of sensitive instructions by the hypervisor. It is possible to identify these differences using a few instructions. An example is CPUID, which returns both the presence of the hypervisor and the hypervisor brand if present.

4. **Table Descriptors**: in this family, we can find techniques that use instructions to retrieve the addresses of OS Table Descriptors, that are data structure used by Intel x86-family processors in order to define the characteristics of the various memory areas used during program execution, including the base address, the size, and access privileges like executability and writability. If a hypervisor is present usually these addresses change. Thus the malware author used these discrepancies to detect virtualized environments. But nowadays, these techniques are reliable only on single-core machines, so they are not useful with modern systems.

5. **Traps**: this family contains all the techniques that use x86 instructions to leak the presence of a hypervisor, emulator, or debugger throwing an exception in a different way than the Exception Handling family. For example with the typical 'INT 3' instruction which is used by a debugger to set breakpoints. So the malware can detect the presence of debuggers if these exceptions are intercepted.

6. **Timing**: this family contains all the techniques that exploit accurate timestamps to identify analysis systems. For example, discrepancies in the execution time of instructions are clear signs of a virtual environment. Indeed, most analysis systems have a significant impact on performance. So, usually these techniques get a first timestamp, then execute some operations and then they get a second timestamp, if the difference between the two timestamps is greater than how expected they detect that are in-

side a virtual environment.

7. **Stalling**: in this family, we can find techniques that use instructions to put the executable in sleep mode to avoid detection. These techniques work because sandboxes have a limited amount of time allocated for the analysis, so they could miss malicious activities of malware samples that activate after a long period. A reasonable time assigned by Sandboxes ranges from 3 minutes to 10 minutes, thus a safe choice is to start malicious actions after 10 minutes to fool the detection system. This behavior is extremely stealthy. Understanding if a sample is stalling is an undecidable problem and, above all, a malware sample does not stall only using sleeping functions offered by the OS, but can do it also performing useless arithmetic operations, or system calls.

8. **Human Interaction**: this family contains all the techniques that exploit the fact that in an automated system where thousands of malware samples are analyzed inside virtual machines, there is no Human Interaction. A way to detect if a human is using the infected machine or not is for example to check the mouse activity, a malware sample can assume that if the mouse cursor is not moving, the environment is instrumented.

9. **Registry**: in this family, we can find techniques that abuse the Windows Registry, which is a container of valuable system information. The Registry is a hierarchical database that contains interesting aspects such as Services available, Programs Installed, and System information that can be retrieved also using Windows API calls. So the techniques of this family access to Windows Registry in search of virtual machines or debuggers related artifacts such as VirtualBox Guest Additions. We can see an example in Figure 4.

10. **WMI**: in this family, we can find techniques that exploit the Windows Management Instrumentation for evasion. The WMI is a proprietary technology by Microsoft for Enterprise Management of Windows Machine. Using the WMI framework and the corresponding query language WQL, it is possible to query any information about Windows machines configuration and change their settings. So with the right query, malware can

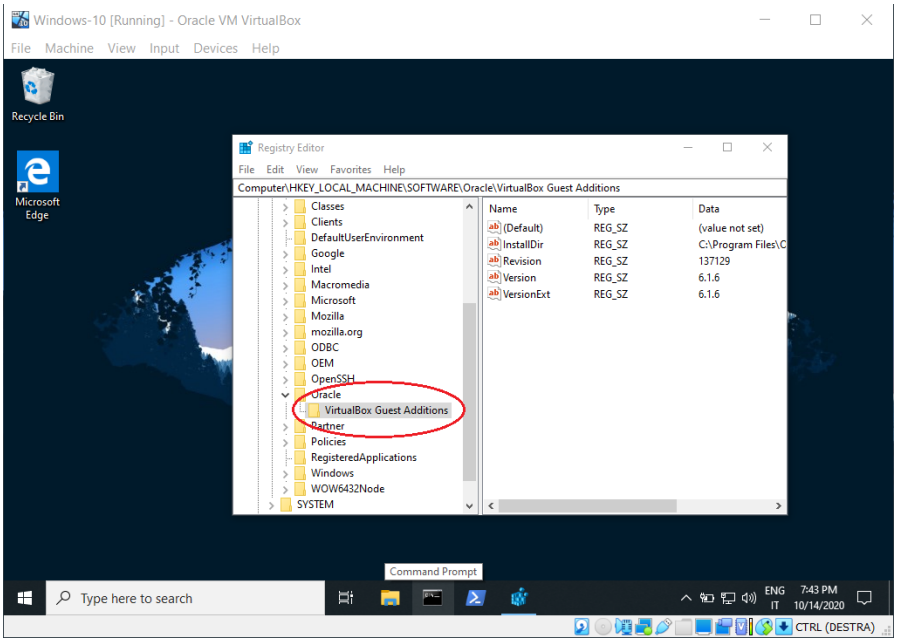
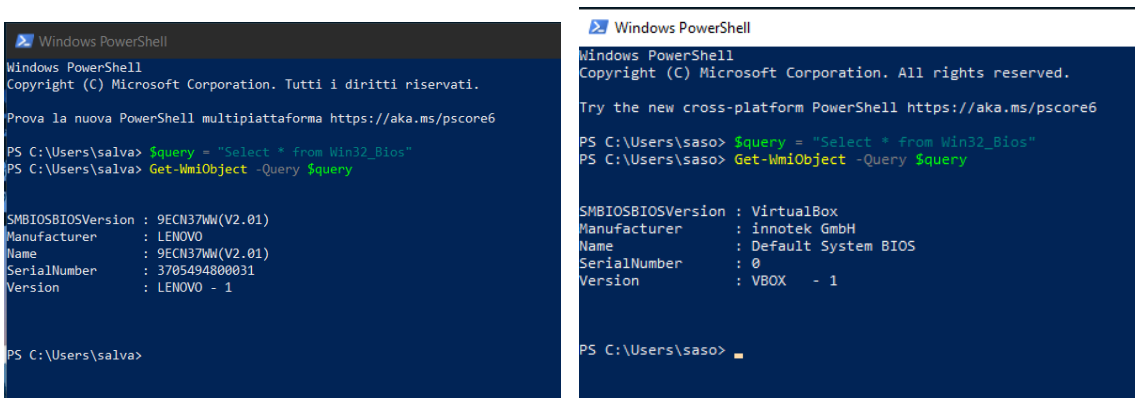


Figure 4: Registry entries usually checked by malware



(a) Normal system (b) Virtual machine

Figure 5: WMI query example

detect if it is inside a virtualized environment. In Figure 5 we can see with an example of a WMI query that retrieves information about the BIOS, the differences in the results between a real environment and one

inside a virtual machine.

11. **System Environment**: this family contains all the techniques that detect system-related information that can spot the presence of a VM or debugger. For example, checking the MAC address that contains a manufacturer fixed part that is well known for virtual machines. Other interesting things to check are the number of processors because usually, VMs have a lower number than normal systems, or the disk size because a virtualized environment uses less space usually.

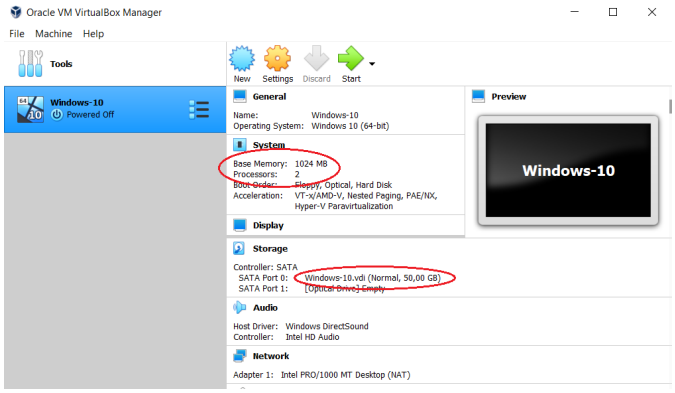


Figure 6: General settings to check for a VM

12. **Process Environment**: in this family, we can find all the techniques that use information about the running process to detect if it is under analysis. For example, a trick used by malware authors to understand if the malware is under debugging is to spawn a child process with debugging privilege and using the Windows API *DebugActiveProcess* try to debug the parent process. Since a process cannot be debugged by two debuggers if the operation fails a debugger is in place.

13. **File System**: this family includes all those techniques that rely on artifacts present on the file system. More precisely malware checks inside known directories if a resource is present or not. For example, in Windows VirtualBox components are usually located under 'C:\ProgramFiles\Oracle\VirtualBox Guest Additions\'. A successful attempt to open this directory is an indi-

cator of VM presence.

14. **List Process**: this family includes all those techniques that list and check the running processes on the system and if they find something suspicious like, for example, 'VBox.exe' they don not act maliciously.

15. **List Services**: in this family, we can find all the techniques that enumerate services. A Windows service is a computer program that operates in the background, it is similar in concept to a Unix daemon. When a Guest OS is installed some hypervisor-related artifacts such as VirtualBox Guest Additions could be needed by the Guest to work properly. Once enumerated all the services, malware search for possible VM components and if one is found they die.

16. **Drivers Information**: this family contains all the techniques that follow the same principle of Services Enumeration, but looking for Drivers. In order to work correctly, most Guest OS contains drivers related to Virtual Peripherals built by the Hypervisor. These components can be enumerated either by direct access to the Windows Driver folder or using a Windows API call.

17. **Others**: this family contains all the remaining techniques we found in similar works that cannot be easily associated with one of the precedent families or that do not produce always reliable results.

18. **Emulation Software**: this family uses a mix of different techniques belonging to the precedent families to detect the presence of emulation software, like QEMU, a free and open-source emulator and virtualizer that can perform hardware virtualization or like Wine a free and open-source compatibility layer that aims to allow application software developed for Microsoft Windows to run on Unix-like operating systems.

2.2 MALWARE ANALYSIS TECHNIQUES

Now that we have more information about all the tricks used by malware authors and about how they try to do evasion, we can see in detail how security

specialists try to defeat all these techniques and the tools they use.

How we said before the first and the most basic way to detect malware is to use a signature-based detection method; with this approach antiviruses and anti-malware analyze a file and compute its hash, if it matches with one inside a database of known malware signature then the file is considered dangerous.

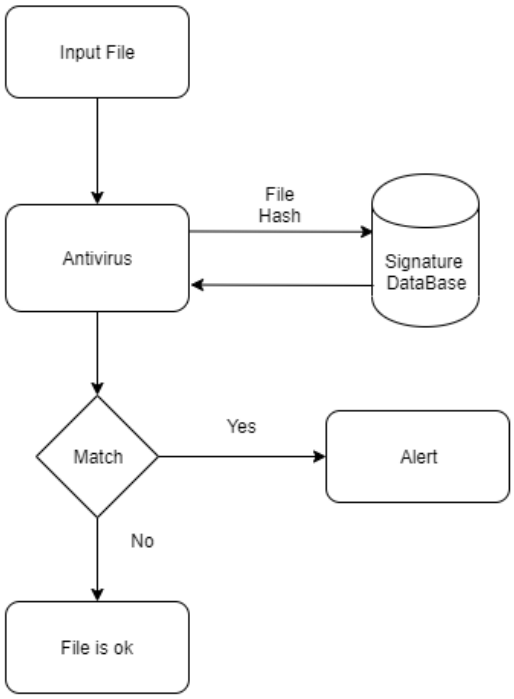


Figure 7: Signature based detection scheme.

The database can contain also code patterns of known malware. In this case the analysis tool checks for a match at byte-level.

This method belongs to a branch of malware analysis called static malware analysis: known also as code analysis because is usually performed by dissecting the different resources of the binary file without executing it and studying each component.

This kind of analysis is usually the first thing that security specialists do when he has to analyze a suspicious file. Understanding what a binary does without running it can be a complicated operation. But to help with the analysis there are different commands and tools that can be used. For example one of the

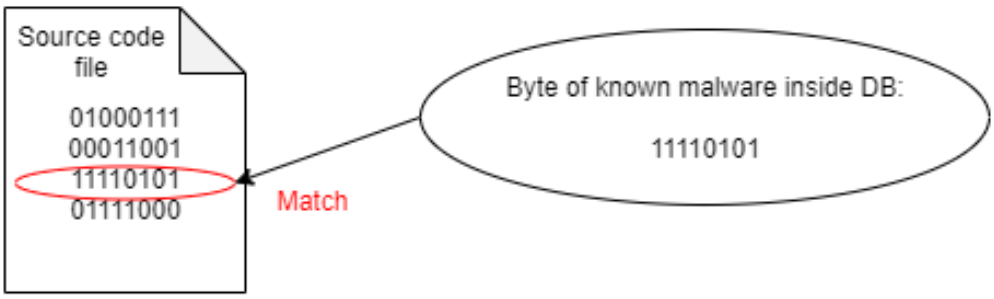


Figure 8: Byte-level match.

most basic command that can be used is the 'strings' one that shows the text inside a binary or data file, this could be useful to obtain, sometimes, clues on what the file does.

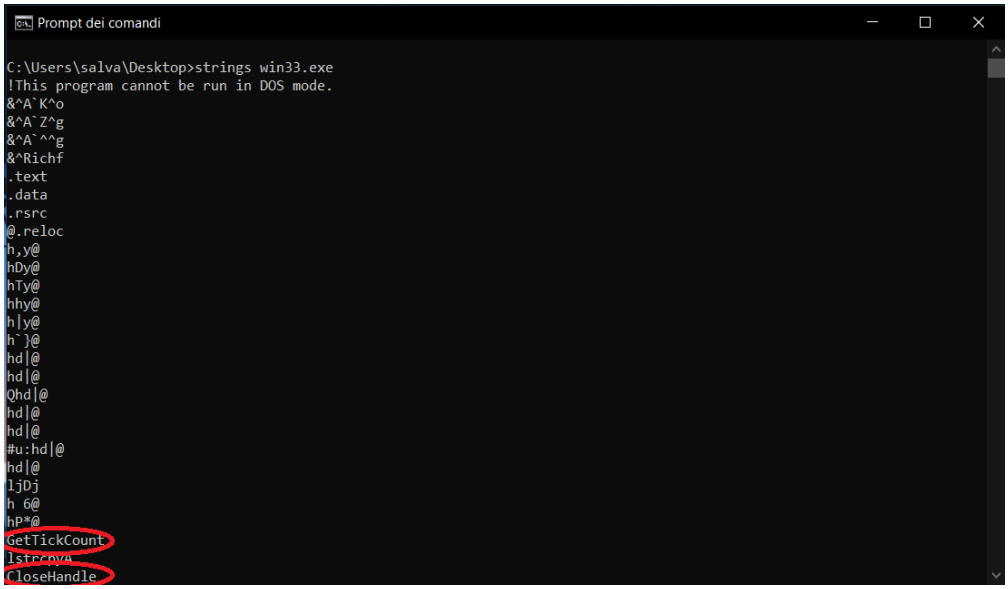


Figure 9: Output of 'strings' command.

For example in Figure 9 we can see that using the 'strings' command against a malware sample we can notice the name of some APIs that usually are used to do evasion.

However simple commands as 'strings' usually with complex malware are not enough. So the malware analysts use Disassemblers tools like Ghidra developed by the NSA, or IDA Pro that is de facto standard to translate the

machine code contained in the binary into assembly code which can be read and understood easily, we can see an example in Figure 10.

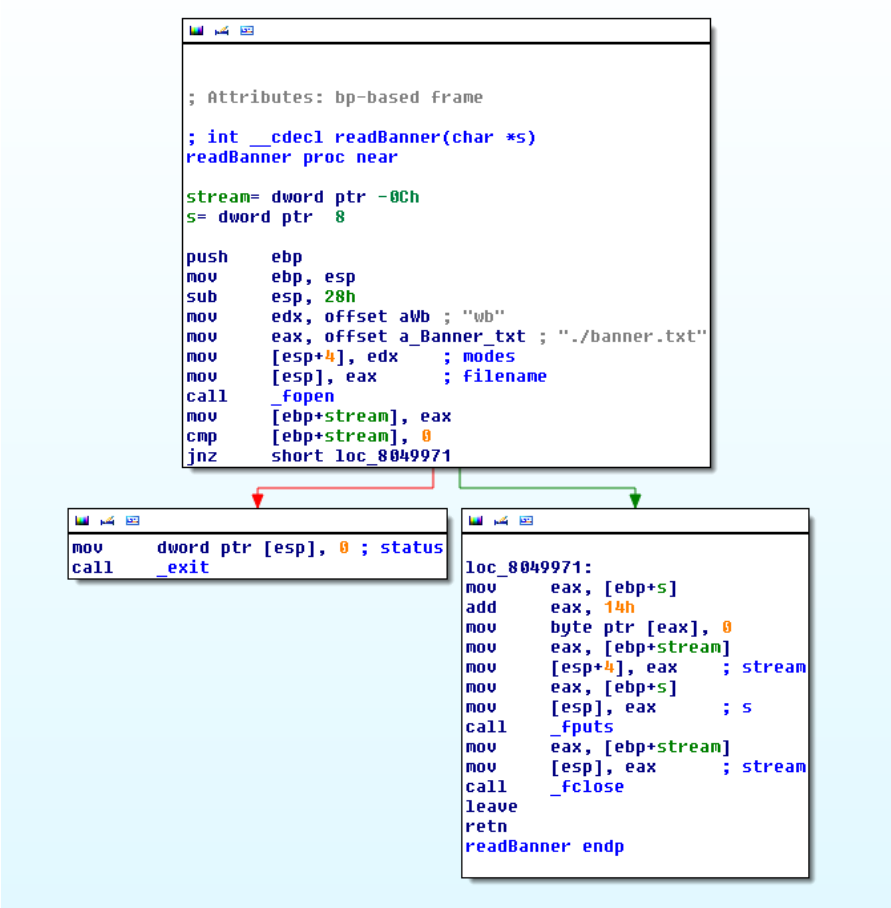


Figure 10: Disassembled code example.

Once that the code is translated into assembly malware analysts can read it and identify specific functions and actions inside the program to have a better point of view on what the program is going to do and how it was originally designed. This reverse engineering operation so helps to find out hidden actions or unintended functionality of the analyzed binaries, but modern malware use evasive techniques to defeat this type of analysis, for example by embedding syntactic code errors that will confuse disassemblers so in the end to analyze the real behavior of the malware, it is necessary to execute it.

This led to the second type of malware analysis, the so-called Dynamic malware analysis or Behavioral analysis that is performed by observing the behavior of the malware while it is actually running on a host system but to avoid that the malware damage the real system of the analyst this form of analysis is often performed in a sandbox environment, many of such sandboxes are virtual systems that can easily be rolled back to a clean state after the analysis is complete. To obtain more information during the execution of a malware sample it is useful also to debug it while is running with a debugger, for example, GDB for Unix-like systems or WinDbg for Microsoft Windows operating systems. This is done to watch the behavior and effects on the host system of the malware step by step while its instructions are being processed.

2.3 SANDBOXING & VIRTUALIZATION

In cybersecurity, a sandbox is an isolated, separated, and restricted environment, usually with limited permissions and limited access to resources, that mimics end-user operating environments. Sandboxes are used to safely execute suspicious code without risking harm to the host device or network.

Just like in a real playground where children can play in the sandbox but are not allowed to play anywhere outside of the sandbox. The limitation of resources (for example of file descriptors, memory, file system space, etc.) is fundamental for security because in case malware is executed the damages will be delimited to the controlled resources providing so another layer of protection against zero-day or high-evasive malware and avoiding them from spreading in all the system. Automated malware analysis systems usually use sandboxes in combination with virtual machines to easily allow to restore a clean environment, saved before the execution of the analyzed file. While sandboxing allows us to execute a program in an isolated, but equal to the host system, environment, Virtualization provides a separate environment within a computer that can function independently from all other environments on the computer. So, a sandbox just isolates a part of the original system, while Virtualization emulates a different environment, for example, we can use a UNIX system as 'guest' on a Windows system we will see as 'host'. So it is important to understand the differences between a sandbox and a virtual machine, when you run an application in a sandbox, it has access to run as if it were not in a sandbox. Anything the application attempts to create or change, however, is lost when the application stops running. In a virtual machine, on the contrary, anything

created or changed by the application is allowed, and everything that happens stays within the virtual machine space.

2.4 STATE-OF-THE-ART OF MALWARE ANALYSIS

In this section we present similar and previous works, to understand which is the actual state-of-the-art for what regards automated malware analysis tools. One of the first tools, similar to our project, used to detect sandboxes and analysis environments is the open-source project 'Paranoid fish' also known as Pafish [37]. As in our work Pafish project employs several evasion-techniques to detect malware analysis tools in the same way as malware does. Another project that uses malware tricks to test anti-malware systems is 'al-khaser' by LordNoteworthy [20], it performs more common malware tricks with respect of Pafish grouped by which tools they are trying to test. Another similar project is the 'ShowStopper - Anti-Debug tricks exploration tool' based on the 'Anti-Debug Tricks' [15] a tool useful for malware researchers to explore and test anti-debug techniques or verify debugger plugins. For what regards the state-of-art of security sandboxes we know that some of them are based on Cuckoo [5] that is an open-source automated sandbox, written in Python, that through Oracle VM VirtualBox, it executes the malware samples. Cuckoo also supports custom-written packages so it is important to test the different sandboxes based on it because the authors could have freely implemented additional features. Another commonly used tool is YARA [36] that is an open-source project used to classify malware samples based on textual or binary patterns. From these patterns, researchers write a description of malware families called rules. For what regard malware analysis an evaluation of automated static analysis tools is provided by Namanya et al. [25], it is a valid work that provides good results, but they focus only on static analysis tools. While our approach focuses above all on dynamic analysis. Egele et al. [9] proposed a survey on automated dynamic malware analysis providing an overview of techniques to analyze unknown and potentially malicious software and of the existing approaches and tools that make use of the introduced techniques. They provide a lot of different and useful information, but the work is almost fifteen years ago, so some information is outdated or tools are not available anymore. Another important evaluation about sandboxing was introduced by Al Ameiri et al. [1], they provide a deep explanation of sandbox implementation focusing on the different types of available sandboxes and their performance, but they do not focus, as we do, on the anti-evasion techniques implemented.

2.5 GOALS

In this thesis, we aim to discover all the countermeasures implemented in automated malware analysis tools against the evasion-techniques used by evasive malware. Taking into account that test all the possible evasion techniques is not feasible due also to the fact that malware authors come up every day with new possible tricks to do evasion, our objective is to collect and test the most known and used in malware samples. We want to test them against the automated malware analysis tools used nowadays by cybersecurity specialists and simple users to evaluate their strength and efficiency against evasive malware. We also aim to exploit more evasion techniques of the available state-of-art tools, dividing them on where or how they try to get the information to understand if the malware is under analysis. To conduct our tests we aim to provide a tool that allows us to add to any executable the evasion techniques that we collected or developed. Then with the obtained results, we want to provide a classification of the most effective evasion techniques and of the anti-evasion techniques implemented in the automated malware analysis tools, above all on the ones freely available on the Internet that use a sandbox, since we did not find a similar classification on previous works, with the hope to provide to everyone valid results that could be useful against evasive malware.

3

APPROACH AND IMPLEMENTATION DETAILS

In this chapter we detail and formalize the approach on which this thesis is based to obtain the wanted results, providing also the implementation details of the different parts of our work, in such a way that everyone could obtain the same results following the same steps. In particular, we provide more information about our implementation of the script, *crafter.py*, that crafts the new sample with the selected techniques added and we also show how we implemented the different evasion-techniques used by evasive malware that we described in the previous chapter.

3.1 APPROACH OVERVIEW

To detect how automated malware analysis tools developers try to defeat the different *evasion techniques* we decided to analyze with these tools a malware which uses evasion techniques. To obtain such malware we added, through our crafter script, the evasion techniques we want to test. If the malware is detected then we suppose that there is an *anti-evasion technique* in that specific tool that nullifies the attempt of doing evasion, or the used evasion-technique does not work anymore, otherwise, the evasion technique does evasion properly.

In Figure 11 we represented with a flow chart our approach to detect the implemented *anti-evasion techniques* in all the different tools for dynamic malware analysis we analyzed. We tested these tools in black with a black-box approach, which is a method of software testing that examines the functionality of an application without having specific knowledge of the application's code or of the internal structure. We are aware of what the software is supposed to do but we do not know how it does it. So, providing an input we base our results only on the output that we receive. While the crafter is the part of our project that generates the samples used to test the tools mentioned before: from any malware builds a new executable which contains any evasion techniques we want and the original malware sample.

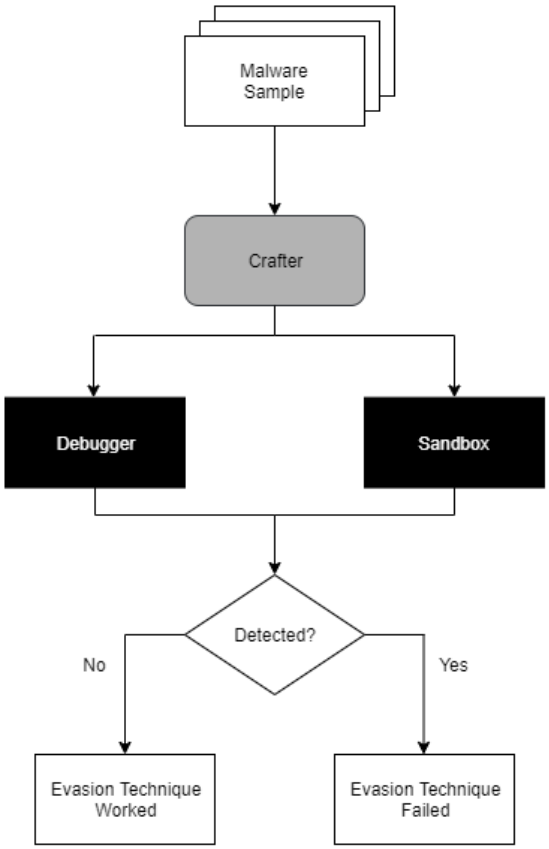


Figure 11: General scheme of our approach to detect anti-evasion techniques

3.2 THE CRAFTER

The crafter is an essential part of our project. As its name suggests, it has the goal to craft the samples we will use to analyze the different automated malware analysis tools. So, it is the part of our project that will make us decide which evasion techniques will be added to our original malware sample. Thanks to the crafter we can generate every time a different sample with different evasion techniques added or with a different malware sample embedded.

As we said before the aim of the crafter is to help us to add any evasion technique we want to any executable. It consists of a python3 script that essentially modifies the Makefile of our project, because we decide the different evasion techniques to use at compile-time, through the use of compilation flags. The main concept of the script can be seen in Figure 12. We provide our malware

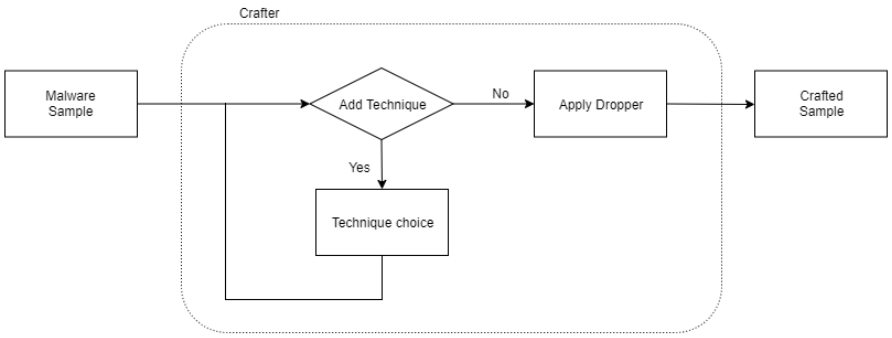


Figure 12: General scheme of *crafter.py*

sample as an argument of the script, then through a loop, we can add all the technique we want, at this point the script embeds [18] the malware sample binary data in a new executable that first of all execute all the evasion techniques we added to it, then if an analysis environment is detected the program terminates, otherwise drops and executes the provided malware sample. So our obtained .exe file acts as a dropper, which is a kind of Trojan designed to "install" some sort of malware (virus, backdoor, etc.) to a target system. There are different ways to drop the malware, its source code can be contained within the dropper itself (single-stage, like in our case) this helps to avoid detection by virus scanners or the dropper may download the malware to the target machine once activated.

In the following listing there is the function that generates the new executable:

```
466 def generateExe():
467     subprocess.run(['make', '-f' ,'Makefile.linux', 'clean'])
468     subprocess.run(['i686-w64-mingw32-ld', '-r', '-b', 'binary', '-o', './output/
        objects/binary.o', 'tmpExe'])
469     subprocess.run(['rm', 'tmpExe'])
470     modifyMake()
471     subprocess.run(['make', '-f' ,'Makefile.linux'])
```

Listing 2: Code to generate the new executable in *crafter.py*

First of all, we create a new object from the malware sample we provided as an argument, that will be embedded inside the new executable, then with the function *modifyMake* we apply all the compilation flags relative to the evasion-techniques we chose, during the execution of the script, modifying the Makefile of our new executable and in the end, we run the make command to generate the new executable.

3.3 THE CRAFTED SAMPLE

In this section, we show how the new executable, that we obtain after the execution of the crafter script, is composed. This executable is the main part of our project, indeed it is the sample that will be analyzed by the sandboxes. It has the goal to execute all the different evasion techniques we decided to test and if all of them fail to detect an analysis environment it has the goal to execute the embedded malware sample.

All the files used to obtain the new executable are written in C and they can be built with MinGW [22], that is a free and open-source software development environment to create Microsoft Windows PE applications, basically is a Windows port of the GNU compiler tools, such as GCC, Make, Bash, and so on and an alternative to the Microsoft Visual C++ compiler and its associated linking/make tools.
The goal of the main file of this new executable is to execute the different evasion techniques we selected before, we implemented all of them in such a way that they return 0 if an analysis environment is not detected, otherwise they return 1. So we maintain these return values inside the *flag* variable and if at the end it still is equal to 0, we have to drop the original malware sample and we do it through the function *doBadStuff* that simply generates a new file called 'embedded.exe' where the malware code will be written and then it executes it with the *system* function.

```
...
extern char binary_tmpExe_start[];
extern char binary_tmpExe_end[];
void doBadStuff(){
  FILE  *fp;
  fp = fopen("embedded.exe", "ab");
  if (fp == NULL){
    puts("Error while opening file");
    exit(1);
  }

  for (char* p = binary_tmpExe_start; p != binary_tmpExe_end; ++p) {
      fprintf(fp,"%c",*p);
  }

  fclose(fp);
  system("embedded.exe");
  getchar();
}
```

```
20
21 int main(void){
22   int flag = FALSE;
23
24   ...
25
26   #ifdef MEMFING
27     printf("[-] Memory Fingerprinting:\n");
28     flag = flag | memoryFingerprinting();
29     printf("\n");
30   #endif
31
32   ...
33   #ifdef EM
34     printf("[-] Looking for emulation softwares:\n");
35     flag = flag | emulation();
36     printf("\n");
37   #endif
38
39   if (flag == FALSE){
40     print_traced("I'm running the program:\n\n");
41     doBadStuff();
42   }
43   else{
44     print_not_traced("We are under analysis.\n\n");
45     getchar();
46   }
47   return 0;
48 }
```

Listing 3: *main.c* source code

We defined all the evasion technique families in the *checks.h* header and its implementation works similarly to the main function, we decide which of the evasion techniques of a family is going to be executed, always through compilation flags that we applied in the Makefile with our crafter script.

```
1 int memoryFingerprinting(){
2   int a = FALSE;
3   int b = FALSE;
4   int c = FALSE;
5   int d = FALSE;
6   int e = FALSE;
7
8   #ifdef MF1
9     a = exec_check("PEB->IsDebugged", &PEB_isDebugged);
10  #endif
```

```
11
12    #ifdef MF2
13      b = exec_check("PEB->NtGlobalFlag", &PEB_NtGlobalFlag);
14    #endif
15
16    #ifdef MF3
17      c = exec_check("PEB->Heap->Flags", &PEB_Heap_Flags);
18    #endif
19
20    #ifdef MF4
21      d = exec_check("PEB->Heap->ForceFlags", &PEB_Heap_ForceFlags);
22    #endif
23
24    #ifdef MF5
25      e = exec_check("KUSER_SHARED->KdDebuggerEnabled", &
        KUSER_SHARED_KdDebuggerEnabled);
26    #endif
27
28    return (a | b | c | d | e);
29  }
```

Listing 4: *checks.c* source code

How we can see from Listing 4 the evasion techniques are executed with the help of the *exec_check* function that needs as parameters a string that describes the evasion techniques we are going to execute and a function pointer to the concrete implementation of the evasion techniques that we will explain in details in the next section.

3.4 EVASION TECHNIQUE FAMILIES

For what regards the implementation of the different evasion techniques we wrote a file, containing the concrete implementation, for each family and we used as references previous works on evasive malware or, in case the evasion technique was described in previous works only theoretically, we used the Microsoft Windows Documentation [35] for the Win32 API used by the techniques. After the implementation of a technique, we tested it, first in a normal environment and then in a virtual machine/debugger, to check if it works correctly.

3.4.1 *Memory Fingerprinting*

How we saw before this family of techniques abuses of structures that hold data about the current process, like the Process Environment Block (PEB) or like the KUSER_SHARED_DATA that contains a lot of Windows system information. So we implemented five different techniques that look for traces of debuggers in these structures, all using the *asm* keyword that allowed us to embed assembler instructions within C code. In this way, we accessed directly the wanted structure and we checked the interested bytes, the PEB, since we are in a 32-bit process, is located at FS [1]:[0x30] [38], while the KUSER_SHARED_DATA is always located in user mode at the address 0x7ffe0000 [40]. Once accessed the right point of the structures we saved the value and we checked them, for example, the PEB->NtGlobalFlag is a byte inside the PEB at offset 0x68 and it is 0 when no debugger is in place and 0x70 otherwise.

3.4.2 *Exception Handling*

This family contains all the techniques that abuse the behavior of the program when occurring exceptions or errors. We implemented five different techniques and the main concept behind their implementation is to define an exception handler for that certain exception and then throw it. If the handler is executed we are in a normal environment, otherwise, we are under debugging because debuggers usually intercept exceptions. In general, our exception handlers simply set a global variable relative to that technique equal to 0, so in the end, we can use the flag's value to understand if we are under analysis. The five techniques we used are well known and they can be found in "The Ultimate Anti-Reversing Reference" [10], we just adapted them for our goals, each of them uses a different way to throw the exception, for example with the *NtClose* function, that normally is used to close a Windows HANDLE [2], if we try to close an invalid handle an exception is generated.

1 Segment register that points to the Thread Information Block (TIB). This structure is created by the kernel on thread creation and is used to support OS-related functionalities, services and APIs.

2 A HANDLE is a context-specific unique identifier to access a resource

3.4.3 *CPU Fingerprinting*

In this family, we implemented the techniques that retrieve information from the actual CPU to detect the analysis environment. We implemented five different techniques, two of them use CPUID, an x86 instruction used to retrieve information about the running CPU, passing EAX=1 as input, the return value describes the processor's features while with EAX=40000000 as input we get as return value the virtualization vendor string in EAX, ECX, and EDX [4] then we check the return values against the known value to detect VM environment. To check if we are in a VMware environment we implemented an evasion technique that uses the privileged x86 instruction IN. This instruction is used, with specific parameters, to communicate with the host in a VM environment, so if it fails we can detect that we are in a normal environment [4]. We implemented also a technique that uses the STR instruction that retrieves the segment selector from the task register, because the value returned by this instruction will differ depending on whether it is run on a host or a virtual machine, we can use it to detect a virtualized environment [3]. The last technique of this family uses the SMSW instructions that store the machine status word into the destination operand, and the return value is known for some VMware versions so we can detect if we are inside a VM [8].

3.4.4 *Table Descr.*

In this family we use three instructions to retrieve information from the OS Table Descriptors and check them against known values:

- "sidt" stores the Interrupt Descriptor Table Register (IDTR) content, this technique is also known as Red pill
- "sldt" stores the segment selector from the Local Descriptor Table Register
- "sgdt" stores the Global Descriptor Table Register (GDTR) content.

These techniques are well known and nowadays deprecated because they are not privileged instructions, so with hypervisors, the retrieved values could well be the same inside and outside of the VM. We just used a classical implementation of these techniques that uses the instructions and checks the returned value [3].

3.4.5 *Traps*

From the point of view of the implementation for this family, we worked in the same way that for the Exception Handling family. We implemented a custom handler for a specific exception, then we throw the exceptions with different instructions. If the handler is executed a global flag is set to 0 and we are in a normal environment otherwise we are under analysis. The instructions we used are:

- "INT 3" that throws a BREAKPOINT_EXCEPTION used by debuggers to set breakpoints [10]
- "VPCEXT" that is an illegal instruction that throws an exception if executed outside of a Virtual PC environment [32]
- "POP SS" used by debuggers to skip the next instruction [14]
- "INT 1" is used by debuggers for a SINGLE_STEP exception [7]
- "ICEBP" is an undocumented x86 instruction that triggers SINGLE_STEP exception with opcode 0xF1 [19]
- "INT 2D" is used to raise a breakpoint exception [19]

3.4.6 *Timing*

This family contains all the techniques that exploit accurate timestamps to identify analysis systems, to implement the techniques of this family we followed for all of them the same approach just using different Windows APIs to retrieve the timestamp. First of all, we made a first call of the API to get a timestamp than we executed many times the CPUID instructions, which is a VM Exit instruction, and then we retrieved a second timestamp. If the difference between the first and second timestamp is greater then the pre-computed average execution time of the CPUID instructions we are under analysis. Some example can be found in "The Ultimate Anti-Reversing Reference" [10].

3.4.7 *Stalling*

For this family that abuses the fact that sandboxes spend only a few minutes analyzing each file, we delayed the execution of the malicious payload by exactly ten minutes. To do it we used different Windows APIs from the official

documentation [35], as for example *Sleep* and *waitForSingleObject* or some undocumented functions of Ntdll.dll [3] [26], like for example *NtDelayExecution*.

3.4.8 *Human Interaction*

For what regards the implementation of this family that checks if there is human interaction during the analysis we called a first time the functions *GetCursorPos* and *GetLastInputInfo* that retrieves respectively the position of the mouse cursor and the time of the last input event, then we stalled the execution for a certain amount of time and then we recalled the two functions. If the two return values were the same we considered that we are in an automated analysis environment.

3.4.9 *Registry*

For the implementation of this family that checks for known Registry Keys in the Windows Registry, we collected known values of registry key related to virtual machines [4] and we used four different NTAPI undocumented functions [26] to check if they were present in the sandboxes:

- "NtOpenKey" tries to open a certain registry key, if the function fails we considered that we are not in a virtualized environment
- "NtEnumerateKey" enumerates all the registry key, so we checked if the ones relative to virtual machines were present
- "NtQueryValyeKey" allows to query a certain key, if the function fails we considered that we are in a normal environment
- "NtEnumerateValueKey" enumerates all registry key value, and we checked for known values.

3.4.10 *WMI*

To use the Windows Management Instrumentation (WMI) queries for environment detection and evasion of dynamic analysis and virtualization engines we

3 One of the most important file in the "Microsoft Windows NT" OS family. Ntdll.dll is mostly concerned with system tasks and it includes a number of kernel-mode functions that enable the "Windows Application Programming Interface (API)".

implemented the main function *wmi_check_query* that accepts as a parameter a WMI Query and executes it, we also pass to this function a function pointer to another function that handles the checks to do to detect if we are under analysis. In this way, we generalized the process to check 22 different WMI Queries used to do evasion and we found these queries in previous works like Al-Khaser [20] or from threat researches [24].

3.4.11 *System Environment*

To detect system information related to a VM or a debugger we used different functions: with *NtQuerySystemInformation*, an undocumented syscall from ntdll.dll, and with *GlobalMemoryStatusEx* we checked the memory size because sandboxes are usually configured to take few GB of memory. Others good indicators of VMs, are the disk size and the number of processors, we used the functions *GetDiskFreeSpaceExA* and *DeviceIoControl* to get the disk size and we accessed the PEB at +0x0b8 [38] to retrieve the number of processors, if the disk was less the 60GB or we had 2 or less processors we considered to be under analysis. To check the MAC Addresses, that usually, in VMs, have specific starting bytes depending on the Hypervisor [4] we used the functions *GetAdaptersAdresses* and *GetAdaptersInfo*. To detect the presence of a debugger we used *IsDebuggerPresent*, *CheckRemoteDebuggerPresent*, *NtQuerySystemInformation(0x23)* and *NtQueryObject* after a call to *NtCreateDebugObject* because if a debug object is created and a debugger is present, we should have two debug objects. Another way to detect VMs, debugger or sandboxes is to check the name of windows on the system or of the computer itself and to do it we used the *FindWindow* and *GetComputerName* functions.

3.4.12 *Process Environment*

We checked the presence of a debugger retrieving information from the running process using the system call *NtQueryInformationProcess* passing as parameter 0x07 or 0x1e or 0x1f [14]. To detect a debugger also *NtGetContextThread* and *NtSetContextThread* can be used, with the first function we can check if a debugger used hardware breakpoints, in a positive case the returned values are always different from zero, while the second function is useful to bypass a hardware breakpoint [14]. Another way to detect a debugger that we implemented is to generate a child process with debugging privileges and try to use the *DebugActiveProcess* function on the parent. Since a process cannot

be debugged by two debuggers, if the attempt fails we can say that the parent process is under debugging. While *NtSetInformationThread(ox11)* [10] and *NtCreateThread(HIDE_FROM_DEBUGGER)* [14] are useful instructions to temper with debugging.

3.4.13 *File System, List Process, List Services and Drivers Information*

The main idea behind these families is to check if certain files, processes, services, or drivers are present in the environment, so for what regards the implementation, we proceeded in a similar way for all of them. We collected all the known strings related to VMs or debuggers files, processes, services, or drivers and with different functions, we try to query or open them, if the functions succeeded we detected the analysis environment. Other approaches we used are just to enumerate all of them and compare with the known strings we collected, or to try to create them, in this case, if the function fails, probably they already exist on the system. In Table 1 we can see the list of checked resources by our techniques, all of them are collected by previous works [4] [33] [20].

3.4.14 *Others*

In this section we collected different techniques implemented in similar projects to work with our tool, many of them are not very powerful evasion techniques and usually, they provide unreliable results. For example with *CanOpenCsrss* we attempt to open csrss.exe if it fails we are not being debugged however if it is successful we 'probably' are [31] or with *hideDesktop* we simply switch the current desktop with a different one with no obvious way to switch back to the old, making difficult the debugging process [15]. *IsParenExplorerExe* checks if the program is executed from the command line, really common in automated malware analysis tools, or clicking the file icon. With *pirated_Windows* or *query_License_Value* we check if the program is running with a valid Windows license or not because usually VMs and sandboxes use not activated versions. The 'VirtualAlloc_WriteWatch' techniques use the MEM_WRITE_WATCH feature of VirtualAlloc to test for additional memory writes by debuggers and sandboxing [20].

Table 1: OS resources to check

File system		
• C:\windows\System32\Drivers:	VmGuestLibJava.dll	vboxoglerrorspu.dll
Vmmouse.sys	VBoxMouse.sys	vboxoglfeedbackspu.dll
vm3dgl.dll	VBoxGuest.sys	vboxoglpackspu.dll
vmdum.dll	VBoxSF.sys	vboxoglpassthroughspu.dll
vm3dver.dll	VBoxVideo.sys	vboxservice.exe
vmtray.dll		vboxtray.exe
VMToolsHook.dll	• C:\windows\System32:	VBoxControl.exe
vmmousever.dll	vboxdisp.dll	vboxoglcrutil.dll
vmhgfs.dll	vboxhook.dll	vboxogl.dll
vmGuestLib.dll	vboxmrxnp.dll	vboxoglarrayspu.dll
List Process		
VGAuthService.exe	ImmunityDebugger.exe	filemon.exe
vmacthlp.exe	Wireshark.exe	procmon.exe
vmtoolsd.exe	dumpcap.exe	regmon.exe
VBoxService.exe	HookExplorer.exe	procexp.exe
VBoxTray.exe	ImportREC.exe	idaq.exe
ollydbg.exe	PETools.exe	idaq64.exe
ProcessHacker.exe	LordPE.exe	httpdebugger.exe
tcpview.exe	SysInspector.exe	Fiddler.exe
autoruns.exe	proc_alyzer.exe	joeboxcontrol.exe
autorunsc.exe	sysAnalyzer.exe	joeboxserver.exe
x64dbg.exe	sniff_t.exe	joeboxserver.exe
x32dbg.exe	windbg.exe	ResourceHacker.exe
List Services		
VMTools	Vmvss	Vmrawdsk
Vmhgfs	Vmscsi	Vmusbmouse
VMMEMCTL	Vmxnet	VBoxService
Vmmouse	vmx_ga	Vmware Tools
Vmware Physical Disk Helper Service		
Drivers Information		
vmnetadapter.sys	vmmouse.sys	vmhgfs.sys
VBoxNetLwf.sys	vmrawdsk.sys	vmmemctl.sys
VBoxUSBMon.sys	VBoxMouse.sys	VBoxSF.sys
VBoxDrv.sys	VBoxGuest.sys	VBoxVideo.sys

Table 2: Processes and Registry Keys for emulation software

	Process	Registry Keys
Virtual PC	VMSrvc.exe VMUSrvc.exe	SOFTWARE\Microsoft\Virtual Machine\Guest\Parameters
QEMU	qemu-ga.exe	HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0 HARDWARE\Description\System
Xen	xenservice.exe	-
Wine	-	SOFTWARE\Wine

3.4.15 *Emulation software*

With these techniques we checked for register keys for different emulation software using the *RegQueryValueEx* and *RegOpenKeyEx* functions. We looked also for processes generated by this emulation software, using a function *GetProcessIdFromName* from the List Process family. In Table 2 we can find all the strings checked by these methods.

4

EXPERIMENTAL VALIDATION AND RESULTS

In this chapter, we explain the experimental validation of our work. First, we formalize and list the goals of the experimental validation. After that, we describe how we setup our tests and which tools we used during our experimental validation. Finally, we present and discuss the results, divided by the different automated malware analysis tools we tested and we provide an evaluation of notable behaviors discovered through the examination of the results of the different tools. We present also an evaluation of the strengths and weaknesses of the different evasion techniques we used to obtain the results, classifying them with regard to their efficacy against the tested tools.

4.1 VALIDATION GOALS AND CHALLENGES

In our experimental validation, we have several goals: first, we want to provide an evaluation to every one of the actual available automated malware analysis tools against the modern evasive malware. We want also to present this evaluation, to everyone is interested, in a good, quick and readable way. Another goal is to classify the used methods to nullify evasion techniques implemented in these tools, obtaining so a clear picture of what has been done so far by sandbox and debugger developers and what should be improved or implemented against evasive malware. We want also to provide a simple but effective tool to test the resistance of the different automated malware analysis systems against evasion techniques, and that this tool covers as many techniques as possible. As final goal, thanks to our tool we want to classify the evasion techniques on the base of their strengths and weaknesses in such a way that security specialists can focus their efforts on defeat the more efficient ones. Our major challenges

are fundamentally to provide feature details of the software that we do not know the internal developments and so to develop a method to understand them from the outside. We have to face also difficulties due to the limitations of the free versions of the available online sandboxes many of them provide partial reports or do not allow some additional features that could be obtained by premium versions.

4.2 EXPERIMENTAL SETUP

To conduct our experiments first of all we generated a crafted executable for each evasion technique we wanted to test, we proceeded in this way to face the sandbox services that do not provide a rich report and to avoid that may be a result from a certain technique affects another one. We decided to use as embedded malware a sample of theZoo repository [39] with a score of 61/70 on VirusTotal [34]. We also tested the original sample against the different sandboxes and it was classified as malicious by each of them. Due to the variety of the different automated malware analysis tools we tested, we did not use a constant setup, but we adapted it with regard to the category of tools analyzed. For example, for what regards to the free available online sandboxes we used a python script to automate the sending process of the crafted sample to all the fifteen sandboxes we tested simultaneously. Such script works thanks to the selenium python package [29] that is used to automate web browser interaction, while for what regard the obtained reports, we analyzed them manually because some of them provided screenshots of the execution so it was easy to understand what was happening inside the sandbox, while for those that did not provide them we checked if the "embedded.exe" file was executed, in that case, we considered the evasion techniques defeated. Instead, for what regard the debuggers we installed them locally on a clean machine and we made the testing phase manually.

4.3 RESULTS

In the following sections, we show the obtained results related to public online available sandboxes and debuggers.

Due to a large number of obtained results, to synthesize, we collected them in tables and we tried to schematize them in a readable way through the use of graphs.In the following tables, we can see the results obtained for each technique, we present overall results with the percentages just to provide a quick overview of the strength of each technique. Looking at the two columns 'Sandbox' and 'Debugger' we can immediately understand the nature of a techniques' family. For example, the techniques of the 'Exception Handling' family work against almost all the debuggers, while against sandboxes they are useless, this shows us that they are intended for anti-debugging purposes, instead, the techniques of the 'Stalling' family are very powerful against sandboxes but they do not work against debuggers, because it is sufficient to skip the instruction, this shows clearly that they are anti-VM techniques.

Table 3: Results grouped by evasion-technique

	Technique	Sandbox(%)	Debugger(%)	Tot(%)
Memory Fingerprinting	PEB->IsDebugged	0/15 (0.0)	5/5(100)	5/20(25)
	KUSER_SHARED->KdDebuggerEnabled	0/15 (0.0)	0/5 (0.0)	0/20(0.0)
	PEB->NtGlobalFlag	0/15 (0.0)	5/5(100)	5/20(25)
	PEB->Heap->Flags	1/15 (6.67)	5/5(100)	6/20(30)
	PEB->Heap->ForceFlags	1/15 (6.67)	5/5(100)	6/20(30)
Exception Handling	SetUnhandledExceptionFilter	1/15 (6.67)	5/5(100)	6/20(30)
	NtClose(INVALID_HANDLE)	0/15 (0.0)	2/5 (40)	2/20(10)
	OutputDebugString	0/15 (0.0)	5/5(100)	5/20(25)
	POPF/D - TRAP FLAG	0/15 (0.0)	5/5(100)	5/20(25)
	CTRL Exception Handling	0/15 (0.0)	4/5(80)	4/20(20)
CPU Fingerprinting	CPUID (EAX=0x00000001)	1/15 (6.67)	0/5(0)	1/20(5)
	IN	0/15 (0.0)	3/5 (60)	3/20(15)
	CPUID (EAX=0x40000000)	1/15 (6.67)	1/5(20)	2/20(10)
	STR	0/15 (0.0)	1/5(20)	1/20(5)
	SMSW	1/15 (6.67)	1/5(20)	2/20(10)
Table Descr.	SLDT	0/15 (0.0)	1/5(20)	1/20(5)
	SGDT	0/15 (0.0)	1/5(20)	1/20(5)
	SIDT	0/15 (0.0)	1/5(20)	1/20(5)
Traps	INT 3	1/15(6.67)	5/5(100)	6/20(30)
	VPCEXT	0/15(0.0)	5/5(100)	5/20(25)
	POP SS	0/15(0.0)	0/5(0)	0/20(0.0)
	INT 1	0/15(0.0)	5/5(100)	5/20(25)
	ICEBP	0/15(0.0)	5/5(100)	5/20(25)
	INT 2D	0/15(0.0)	5/5(100)	5/20(25)
Timing	GetTickCount	0/15(0.0)	0/5(0)	0/20(0.0)
	RDTSC/D	11/15(73.3)	0/5(0)	11/20(55)
	QueryPerformanceCounter	13/15(86.6)	0/5(0)	13/20(65)
	GetLocalTime	9/15(60.0)	0/5(0)	9/20(45)
	timeGetTime	5/15(33.3)	0/5(0)	5/20(25)
	NtQuerySystemTime	13/15(86.6)	0/5(0)	13/20(65)
	GetSystemTime	3/15(20)	0/5(0)	3/20(15)
	KUSER_SHARED->SystemTime	11/15(73.3)	0/5(0)	11/20(55)
	KUSER_SHARED->InterruptTime	11/15(73.3)	0/5(0)	11/20(55)
	timeGetSystemTime	6/15(40)	0/5(0)	6/20(30)
	KUSER_SHARED->TickCountQuad	5/15(33.3)	0/5(0)	5/20(25)
	NtGetTickCount	7/15(46.6)	0/5(0)	7/20(35)
	QueryInterruptTime	15/15(100)	0/5(0)	15/20(75)
	NtQueryPerformanceCounter	2/15(13.3)	0/5(0)	2/20(10)
	QueryUnbiasedInterruptTimePrecise	15/15(100)	0/5(0)	15/20(75)
	QueryInterruptTimePrecise	15/15(100)	0/5(0)	15/20(75)
	GetTickCount64	6/15(40)	0/5(0)	6/20(30)
	QueryUnbiasedInterruptTime	7/15(46.6)	0/5(0)	7/20(35)

Table 4: Results grouped by evasion-technique (2)

	Technique	Sandbox(%)	Debugger(%)	Tot(%)
Stalling	waitForSingleObject/Ex	12/15(80)	0/5(0.0)	12/20(60)
	Sleep/SleepEx	5/15(33.3)	0/5(0.0)	5/20(25)
	SetTimer	15/15(100)	0/5(0.0)	15/20(75)
	SetWaitableTimer/Ex	13/15(86.6)	0/5(0.0)	13/20(65)
	CreateTimerQueueTimer	15/15(100)	2/5(40)	17/20(85)
	NtDelayExecution	5/15(33.3)	0/5(0.0)	5/20(25)
	timeSetEvent	12/15(80)	1/5(20)	13/20(65)
	icmpSendEcho/2/Ex	13/15(86.6)	1/5(20)	14/20(70)
HI	GetCursorPos	5/15(33.3)	0/5(0.0)	5/20(25)
	GetLastInputInfo	0/15(0.0)	0/5(0.0)	0/20(0.0)
Registry	NtOpenKey/Ex	0/15(0.0)	0/15(0.0)	0/20(0.0)
	NtEnumerateKey	1/15(6.6)	0/15(0.0)	1/20(5)
	NtQueryValueKey	0/15(0.0)	0/15(0.0)	0/20(0.0)
	NtEnumerateValueKey	0/15(0.0)	0/15(0.0)	0/20(0.0)
WMI (ExecQuery)	Win32_Processor	8/15(53.3)	0/15(0.0)	8/20(40)
	Win32_LogicalDisk	2/15(13.3)	0/15(0.0)	2/20(10)
	Win32_BIOS	2/15(13.3)	0/15(0.0)	2/20(10)
	Win32_ComputerSystem (model)	1/15(6.6)	0/15(0.0)	1/20(5)
	Win32_ComputerSystem (manufacturer)	3/15(20)	0/15(0.0)	3/20(15)
	Win32_ComputerSystem (ProcessorId)	0/15(0.0)	0/15(0.0)	0/20(0.0)
	Win32_Fan	1/15(6.6)	0/15(0.0)	1/20(5)
	Win32_CacheMemory	0/15(0.0)	0/15(0.0)	0/20(0.0)
	Win32_PhysicalMemory	0/15(0.0)	0/15(0.0)	0/20(0.0)
	Win32_MemoryDevice	0/15(0.0)	0/15(0.0)	0/20(0.0)
	Win32_MemoryArray	0/15(0.0)	0/15(0.0)	0/20(0.0)
	Win32_VoltageProbe	0/15(0.0)	0/15(0.0)	0/20(0.0)
	Win32_PortConnector	0/15(0.0)	0/15(0.0)	0/20(0.0)
	Win32_SMBIOSMemory	0/15(0.0)	0/15(0.0)	0/20(0.0)
	Win32_PerfFormattedData_count_therm	0/15(0.0)	0/15(0.0)	0/20(0.0)
	CIM_Memory	0/15(0.0)	0/15(0.0)	0/20(0.0)
	CIM_Sensor	0/15(0.0)	0/15(0.0)	0/20(0.0)
	CIM_NumericSensor	0/15(0.0)	0/15(0.0)	0/20(0.0)
	CIM_TemperatureSensor	0/15(0.0)	0/15(0.0)	0/20(0.0)
	CIM_VoltageSensor	0/15(0.0)	0/15(0.0)	0/20(0.0)
	CIM_PhysicalConnector	0/15(0.0)	0/15(0.0)	0/20(0.0)
	CIM_Slot	0/15(0.0)	0/15(0.0)	0/20(0.0)
System Environment	NtQuerySystemInfo(PHY_MEM_NFO)	10/15(66.6)	0/5(0.0)	10/20(50)
	FindWindow	0/15(0.0)	0/5(0.0)	0/20(0.0)
	GetComputerName	0/15(0.0)	0/5(0.0)	0/20(0.0)
	IsDebuggerPresent	0/15(0.0)	5/5(100)	5/20(25)
	GetAdaptersInfo	1/15(6.6)	0/5(0.0)	1/20(5)
	GetDiskFreeSpace/Ex	1/15(6.6)	0/5(0.0)	1/20(5)
	CheckRemoteDebuggerPresent	0/15(0.0)	5/5(100)	5/20(25)
	GetAdaptersAdresses	1/15(6.6)	0/5(0.0)	1/20(5)
	GlobalMemoryStatusEx	3/15(20)	0/5(0.0)	3/20(15)
	NtQuerySystemInformation(0x23)	2/15(13.3)	0/5(0.0)	2/20(10)
	DeviceIoControl	1/15(6.6)	0/5(0.0)	1/20(5)
	PEB->NumProcessors	1/15(6.6)	0/5(0.0)	1/20(5)
	NtQueryObject(after)NtCreateDebugObject	0/15(0.0)	0/5(0.0)	0/20(0.0)

Table 5: Results grouped by evasion-technique (3)

	Technique	Sandbox(%)	Debugger(%)	Tot(%)
Process Environment	NtSetContextThread(CNTXT_DBG_REGS)	0/15(0.0)	0/5(100)	0/20(0.0)
	NtQueryInformationProcess(0x07)	0/15(0.0)	5/5(100)	5/20(25)
	NtSetInformationThread(0x11)	0/15(0.0)	5/5(100)	5/20(25)
	NtQueryInformationProcess(0x1e)	0/15(0.0)	5/5(100)	5/20(25)
	NtQueryInformationProcess(0x1f)	0/15(0.0)	5/5(100)	5/20(25)
	NtGetContextThread(CNTXT_DBG_REGS)	0/15(0.0)	0/5(0.0)	0/20(0.0)
	DebugActiveProcess on Parent	1/15(6.6)	5/5(100)	6/20(30)
	NtCreateThreadEx(HIDE_FROM_DBG)	1/15(6.6)	0/0(100)	1/20(5)
File System	NtOpenFile	0/15(0.0)	0/0(100)	0/20(0.0)
	NtQueryAttributesFile	0/15(0.0)	0/0(100)	0/20(0.0)
	NtCreateFile	7/15(46.6)	0/0(100)	7/20(35)
	NtQueryDirectoryFileEx	11/15(73.3)	0/0(100)	11/20(55)
List Proc	NtQSI(SYSTEM_PROCESS_INF)	0/15(0.0)	0/0(100)	0/20(0.0)
	EnumProcesses	0/15(0.0)	0/0(100)	0/20(0.0)
	GetModuleBaseName	0/15(0.0)	0/0(100)	0/20(0.0)
	GetProcessIdFromName	1/14(7.14)	3/5(60)	4/19(21)
List Services	OpenSCManager	0/15(0.0)	0/5(0.0)	0/20(0.0)
	EnumServicesStatus	1/15(6.6)	0/5(0.0)	1/20(5)
	OpenService	0/15(0.0)	2/5(0.0)	2/20(10)
	GetServiceDisplayName	0/15(0.0)	0/5(0.0)	0/20(0.0)
	GetServiceKeyName	0/15(0.0)	0/5(0.0)	0/20(0.0)
Driv	EnumDeviceDrivers	1/15(6.6)	0/5(0.0)	1/20(5)
	GetDeviceDriverBaseName	1/15(6.6)	0/5(0.0)	1/20(5)
Others	IsParentExplorerExe	8/15(53.5)	5/5(100)	13/20(65)
	CanOpenCsrss	6/15(40)	1/5(20)	7/20(35)
	MemoryBreakpoints_PageGuard	0/15(0.0)	5/5(100)	5/20(25)
	NtYieldExecution	9/15(60)	0/5(0)	9/20(45)
	SetHandleInformatiom_ProtectedHandle	0/0(0.0)	5/5(100)	0/20(0.0)
	query_License_Value	2/15(13.3)	0/5(0)	2/20(10)
	power_Capabilities	5/15(33.3)	0/5(0)	5/20(25)
	pirated_Windows	4/15(26.6)	0/5(0)	4/20(20)
	check_LoadLibrary	0/15(0.0)	4/5(80)	4/20(20)
	instructionPrefixes	1/15(6.6)	5/5(100)	6/20(30)
	IsHooked	0/15(0.0)	1/5(20)	1/20(5)
	dbgExcp	0/15(0.0)	5/5(100)	5/20(25)
	VirtualAlloc_WriteWatch_BufferOnly	0/15(0.0)	0/5(0.0)	0/20(0.0)
	VirtualAlloc_WriteWatch_APICalls	0/15(0.0)	0/5(0.0)	0/20(0.0)
	VirtualAlloc_WriteWatch_IsDebuggerPresent	0/15(0.0)	0/5(0.0)	0/20(0.0)
	VirtualAlloc_WriteWatch_CodeWrite	0/15(0.0)	5/5(100)	5/20(25)
	hideDesktop	0/15(0.0)	0/5(0.0)	0/20(0.0)
Emulation	Virtual PC	0/15(0.0)	0/5(0.0)	0/20(0.0)
	QEMU	0/15(0.0)	0/5(0.0)	0/20(0.0)
	Xen	0/15(0.0)	0/5(0.0)	0/20(0.0)
	Wine	0/15(0.0)	0/5(0.0)	0/20(0.0)

In the next sections, we will show in detail the results for all the different malware analysis tools we analyzed, showing the data obtained concerning the single tools.

4.3.1 *Public sandboxes results*

First of all, in the following Figure, we can see a general representation of the obtained results for what regard the public online sandboxes.

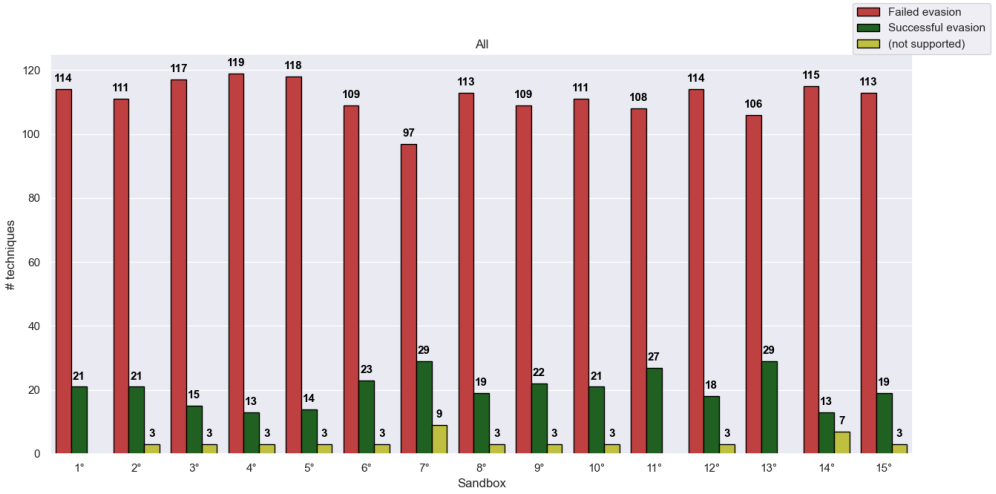


Figure 13: General results of public sandboxes

The red columns "Failed evasion" represent the number of evasion techniques that failed to do evasion in a certain sandbox, while the green columns represent the number of evasion techniques that did evasion with success, making impossible the analysis of the malware sample. For what regards the yellow columns, they represent the number of evasion techniques that make crash the executable in that certain sandbox, usually this is caused by the use of an API too new for the operating system version on which is based the sandbox. More precisely every new Windows version introduces new APIs that are not available in the older ones, so a program developed with APIs introduced in Windows 10, for example, cannot be executed in a machine with Windows XP. This shows immediately that only three online public sandboxes use the last version of the Windows operating system, while the others usually use Windows 7, and some of them still Windows XP. It is important to underline that we separated the count of not supported APIs that make the executable crash from the count of evasion techniques that worked correctly, simply to

show better this behavior, but at the end, they have to be considered as perfectly working evasion techniques because they prevent the malware from the analysis. This can explain also because, as we can see from Tables 3, 4 and 5, there are some evasion techniques intended as anti-debugging tricks that work against sandboxes, indeed if one of these techniques use APIs call not supported by the OS version used in the sandbox the sample will crash and the malware cannot be analyzed. From Figure 13 we can also see that the average number of working evasion techniques is twenty-three, with a range of scores that goes from sixteen up to thirty-eight. It is important also to notice that sandboxes, during their behavior analysis, classify as dangerous a file also if suspicious windows commands are executed or if certain API calls are made. For example, it is sufficient to have a call of the *IsDebuggerPresent* function to obtain a high severity signature that makes our file to be reported as malicious also in the case there is nothing malicious during the execution, except for the API calls that usually are used to do evasion. So during the collection of our results, we did not consider the score assigned by the sandboxes but just if our malicious payload is executed or not.

4.3.2 *Debuggers results*

In this section, we provide and comment the result of all the major debuggers available for Windows Systems.

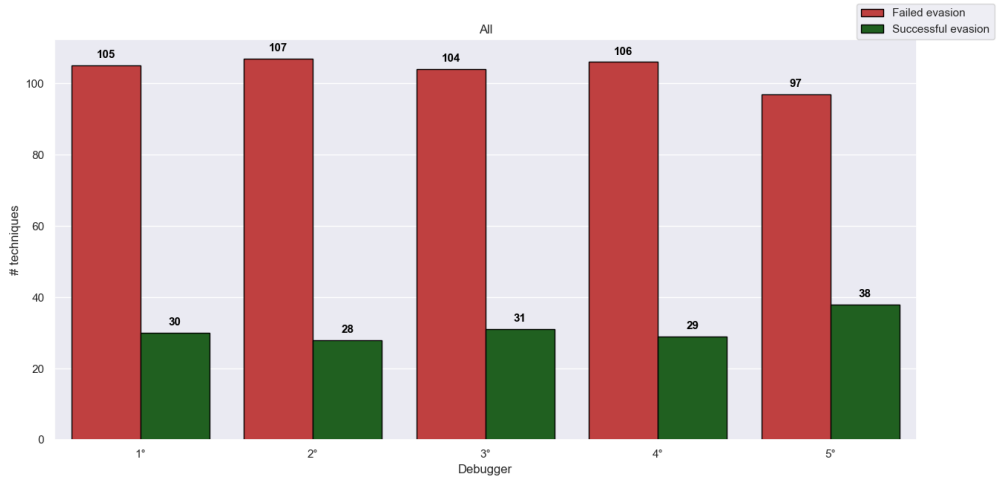


Figure 14: General results of Windows Debuggers

From the graph in Figure 14 we can notice how, on average, all the debuggers have the same "strength" against evasion techniques, that are not so elevated. On average, only thirty techniques over one hundred thirty-five worked, but we have to consider that those that did not work are anti-VM techniques, so each technique, developed specifically to do evasion against debuggers, works fine even if they are really old and well-known. This shows that, for what regard debuggers, no one took countermeasures against evasive malware, this could seem reckless, but considering that debuggers usually are used manually by the security specialists, unexpected behaviors, as a crash of the debugger during the analysis of a sample, are easily detected. Moreover, with single-step execution, each instruction can be seen and evasion techniques can be quickly detected and defeated by a cybersecurity expert. It is interesting to notice from Tables 3, 4 and 5 that some evasion techniques that are intended to work against VMs work also against debuggers, this happens because these techniques usually use privileged instructions that make the debugger crash.

4.4 SANDBOXES ASSESSMENT

In this section, we aim to provide a general assessment of the different sandboxes we tested with respect to the collected results. As we said before of the fifteen online sandboxes we analyzed, only three of them use the last version of the Windows OS, just one of them uses still Windows XP, while all the others use Windows 7. It is interesting to notice that no one uses Windows 8 as an operating system and that the worst score is of the only one that uses the oldest operating system, but newer OS versions do not imply better results indeed two of the sandboxes using Windows 10 got a score greater than the average. During our tests we noticed that at least three online sandboxes are based on Cuckoo, moreover from Tables 3, 4 and 5 we can deduce that no sandboxes apply a debugger during the analysis of a sample indeed all the anti-debugging evasion techniques usually fail, while in the few cases of success is because the program crashed due to the old version of the OS, this makes the Memory Fingerprinting, Exception Handling, Traps and Process Environment almost useless. For what regard the anti-evasion techniques implemented by the developers of these automated malware analysis systems they are focused above all on hiding the presence of the virtualized environment, for example against the lack of human interaction in many of the sandboxes is implemented an automatic movement of the mouse cursor or random input during the execu-

tion of the suspicious file. Another important precaution used is to empty or to hide from the Windows Registry every known key value related to the virtualized environment, indeed all the techniques belonging to the Registry family failed against almost all the sandboxes. We can hypothesize also that the developers of these online sandboxes denied the access or anyway hide in some way, as we already saw for the Windows Registry, all the sensible resources to the identification of the system indeed all the techniques based on listing files, directories, processes, services, and drivers have a really low success rate. Another interesting aspect to notice is from the System Environment family: from Table 4 we can notice that *GlobalMemoryStatusEx*, that retrieves information about the system's current usage of both physical and virtual memory, worked only against three sandboxes over fifteen, but the *NtQuerySystemInformation* that, with the right parameter, it is used fundamentally for the same purpose and it is implemented to check the same size of memory, worked against ten sandboxes, this can be explained by the possible presence of a Windows API Hooker that detects and apply the countermeasures of evasive techniques based on system calls and Windows APIs, in this case incrementing the return value of available memory. The presence of a possible x86 Instructions Monitor is confirmed also, for example, with the result got with the CPUID instruction from the CPU Fingerprinting family that it should return values describing the processors' features, but it works only against the weakest sandboxes, but using the *Win32_Processor* WMI query that retrieves similar CPU information we have better results. We found other evidence of the presence of such countermeasures thanks to the Stalling family, that is the most powerful one against sandboxes, indeed the timer that we set in different ways sometimes was nullified by the sandboxes, it is the case of the *Sleep* function, but to be more precise we know that internally this function executes the syscall *NtDelayExecution* and in fact, they have the same success rate. A separate discussion must be made for the techniques belonging to the Timing family, for the ones based on the time elapsed since the system boot, like *KUSER_SHARED->TickCountQuad, GetTickCount, GetTickCount64* and *NtGetTickCount* the value could have been easily crafted because they work but not well as the techniques based on discrepancies in the execution time of instructions, indeed these techniques work very well, above all *QueryInterrupTime, QueryUnbiasedInterruptTimePrecise* and *QueryInterruptTimePrecise* that are functions recently added and so they make crash the execution in all the sandboxes not based on Windows 10. We also noticed, testing many times the same sample, that the results were not always the same for this kind of techniques and we counted a technique as working only if the average of the success rate was quite high, we deduced so that the

actual workload of the automated malware services maybe can influence the performance. This behavior could be considered similar to that of a normal environment, indeed slowdowns are frequent, but testing the evasion techniques also in an everyday laptop we noticed that they rarely provide wrong results. In conclusion, about these public sandboxes, we can say that they provide, generally, a good level of detection against malicious files because also if there are still many working evasion techniques that hide our malware payload from the analysis the reports detected almost always suspicious elements that should alert those who submitted the file for analysis. Despite the excellent countermeasures against all those techniques based on listing files, processes, services, registries, or drivers related to virtualized environments, some improvements should be done for what regards the performance of the sandboxes in such a way to defeat the techniques based on Timing or about the evasion techniques based on Stalling in the same way as it has been done, for example, for the *NtDelayExecution* API, however having tested only free versions of the sandboxes we cannot say that such improvements are not already implemented in the premium versions which add many features for the analysis. All the previous considerations led us to consider online free available sandboxes good tools for a basic malware analysis that can be used also by simple users if they have doubts about some files or if there is a very high number of samples to test, but for advanced analyses, at least for the free versions, they cannot replace the experience of a security expert and they have to be considered as the first step of the analysis process.

4.5 EVASION TECHNIQUES: STRENGTHS AND WEAKNESSES

In this section, we aim to clarify the strengths and weaknesses of the different evasion techniques we tested on the base of the obtained results. From Tables 3, 4 and 5 we can immediately notice that the most effective evasion techniques against sandboxes are, without any doubt, the ones belonging to the Timing and Stalling families.

From Figure 15 and 16 we can see detailed data about every single sandbox, for what regards the Stalling family its strength is to abuse the limited amount of time assigned to the analysis, indeed, on average, each sandbox runs the sample for a maximum of five minutes so if a malware, as in our case, starts to act maliciously after 10 minutes it fools the detection system. Also if the sandboxes' developers as countermeasure try to set to a null value the parameter of known functions used to stall the execution, detecting this kind of evasion

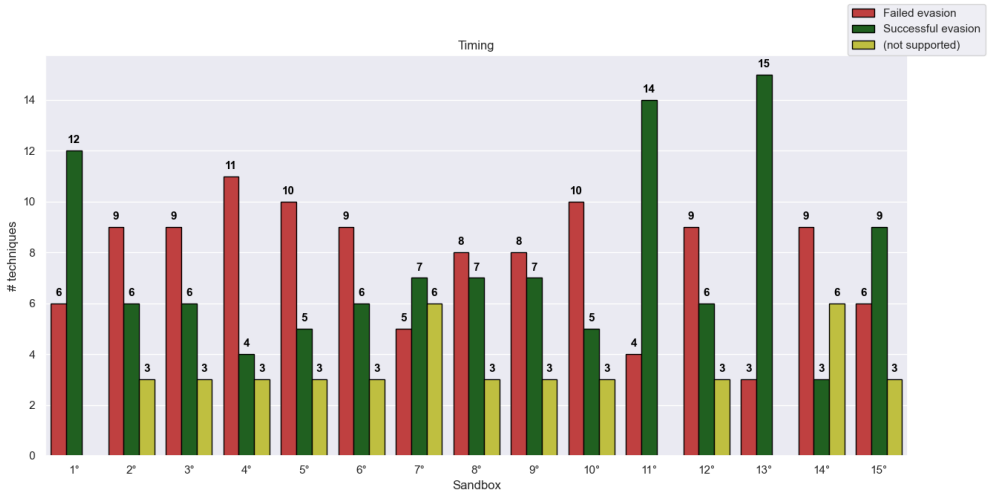


Figure 15: Timing results against online sandboxes

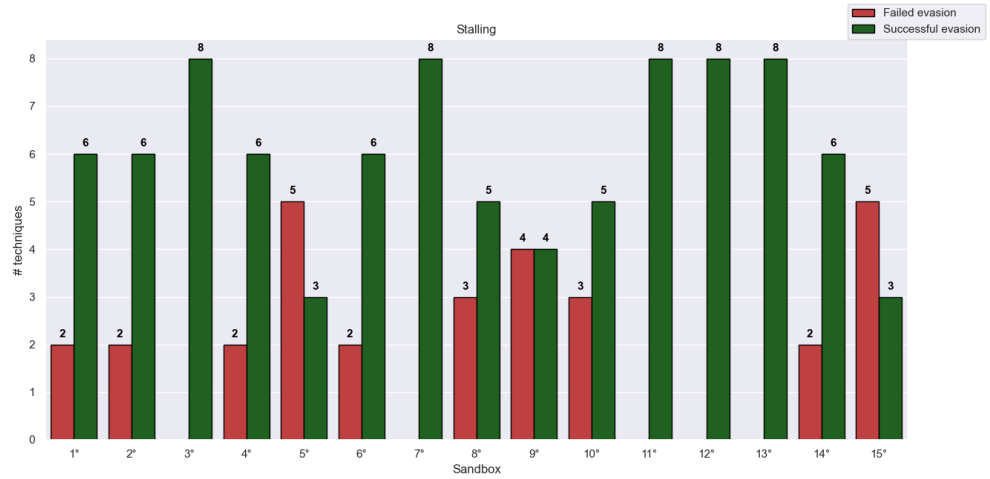


Figure 16: Stalling results against online sandboxes

remains a tough problem, because understanding, if a sample is stalling, is an undecidable problem and a malware can use other techniques beyond the sleeping functions offered by the OS, for example performing useless arithmetic operations [13] [27], while this extremely stealthy behavior could be easily detected by manual analysis and nullified, for example, using a debugger to skip the instructions. On the other side, the strength of the Timing family relies on detecting discrepancies in the execution time of VM Exit instructions such as CPUID between two timestamps and this can be used also against debug-

gers if the analysis is made with single-steps. Another strength of this family is to use many functions added only in recent OS versions and since only three sandboxes use the last version of Windows the most of them is not able to analyze malware that instead works on the 57.21% of laptops [21]. Another family with a useful strength is the WMI one, indeed it is unfeasible to understand from WQL queries which specific information a malware is looking for, and this led to increased use of WMI queries for environment detection and we already saw in the previous section that we retrieved successful CPU information in this way, while classical methods like the use of the CPUID instruction failed.

The major weaknesses regard the techniques based on detecting sandbox artifacts, because the malware use vendor-specific information associated with popular sandbox products, like for example, the existence of certain files, processes, drivers, file system structure, Windows ID, username, etc. to reveal the presence of a virtualized environment, but this information have been known for a long time now and then replace them with random strings is not a hard task for sandbox developers, defeating so completely this kind of techniques.

Other useless techniques are the ones based on STR, SLDT, SGDT, and SIDT instructions, their weakness is related to the fact that they are instructions useful against single-core physical machines that are no longer used. To conclude, for what regards instead all the techniques intended as anti-debugging tricks they work perfectly fine against debuggers, but they have obviously poor results against sandboxes.

5 LIMITATIONS & FUTURE WORKS

This chapter focuses on our project's limits. First, we present the shortcomings in our work. Then, we propose new branches that can be explored to improve our research about automated malware analysis tools.

5.1 LIMITS

So far we have presented our research about anti-evasion techniques implemented in public sandboxes, showing our approach to detect them, highlighting the general context in which we developed a tool to test the actual available automated malware analysis tools, then we presented how we conducted the experiments to obtain the results from which we made our assessments. In this chapter, we want to provide the limits of our tool and of our investigation and we want to propose also some possible improvements for future work related to the same research field.
The first limitation is due to the usage of free versions of the online automated malware analysis tools: inevitably, many important features are accessible only to premium users, and usually, these paid versions are not cheap and rarely affordable for simple users. This caused problems, first, in some cases, we had difficulties to extract the results from the incomplete reports provided to free users, but above all, we could not use fundamental options against evasion, like increasing the duration of the analysis or using, as called by some of the services, heavy anti-evasion analysis. Evasion techniques that did evasion successfully against the free versions we tested, with the use of these premium features, maybe could fail, providing us different results and increasing the resistance of these tools against evasive malware. We also notice that some sandboxes in their free versions use as OS version Windows 7, while with a premium account the user can decide among all the Windows operating systems, this would modify, for example, the results of the Timing family because, as we saw, it has many techniques using Windows 10 APIs which in lower OS versions make the executable crash doing so evasion correctly. Another small limit of our work is due to the fact that many evasion techniques use specific 32-bit instructions our tool generates a 32-bit executable, so we were

unable to add few techniques that work only with 64-bit executables as the *WudfIsAnyDebuggerPresent*, *WudfIsKernelDebuggerPresent* and *WudfIsUserDebuggerPresent* [20], function presents in the WUDFPlatform.dll file, but they are fundamentally anti-debugging tricks, so against the sandboxes, generally, they should not work, while for what regards debuggers, we tested them and they worked fine.
Finally due to our black-box approach during the testing phase, as we explained before we provide the input and we consider only the output without caring about the internal structures of the sandboxes, some of our conclusions about the anti-evasion techniques implemented by the developers could be wrong, being all the sandboxes we tested closed source software this limit cannot be overwhelmed we could only speculate about the presence of countermeasures similar to those present in open source sandbox-like Cuckoo [5].

5.2 FUTURE WORKS

As mentioned in the previous section, there are some aspects of our work that could be improved. We tested 135 different evasion techniques, but we know that there are many others and that everyday malware authors come up with new tricks against analysis, so due to the design of our project it is possible to add them in the future without any effort, it is enough to add their implementations and make little modifications in the already saw *main.c* and *checks.c* files and in our crafter script. This could make our tool even more powerful and useful for testing sandboxes and automated malware analysis tools. A second improvement can be done by studying the premium version of the sandboxes we tested, in this way we could try if, with the use of the premium options, we will obtain different results than the ones we got with the free versions, a similar work it would be also useful to underline the differences between free and paid versions of the actual available automated malware analysis tools. Another possible future work it would be also to use our tool to test the sandboxes used by antiviruses, indeed many antiviruses use sandboxes to analyze suspicious file working in a similar way of the sandboxes we analyzed, they automatically lock unknown files in a secure environment where they can cause no damage and after their execution, they provide a report to the user. To accomplish this task antiviruses also use cloud sandboxes that use artificial intelligence and machine learning technologies to detect never-seen-before malware by studying the behavior of the samples, so it would be very interesting to test their resistance against evasion techniques using our tool.

Finally, it would be interesting to make a different version of our project able to run in other operating systems, like Mac or Linux, to test the evasion techniques used in these environments, indeed Malwarebytes[1] is out with a new report [16] in which it states that Mac malware is growing faster than that for Windows and we know that automated sandboxes that analyze malware targeting Linux and macOS are already available, also in free online versions. The same work could be done also for what regard Android ad iOS operating systems, indeed with the emergence of mobile application markets, there has been a dramatic increase in mobile malware, and mobile platform providers are constantly creating and refining their malware-detection techniques, including static analysis and behavioral monitoring and a similar too to the one we developed could be useful.

1 American Internet security company specialized in protecting home computers, smartphones, and companies from malware and other threats.

6

CONCLUSIONS

As we saw dynamic malware analysis is now considered the last line of defense against advanced threats, but due to the high number of malware samples developed nowadays cybersecurity specialists developed automated malware analysis tools to speed up the analysis process. So evading detection in these analysis environments now is the objective of any malware author, we also saw that during the years they developed many different ways to detect these environments and to conceal the real behavior of their malware and that they are keeping to find every day new ways to do it. In this thesis we collected and developed many of these evasion-techniques, providing a concrete implementation for some of them that until now were only theoretically explained in previous works, and we made a tool capable of applying them to any executable we desire. Armed with our tool we decided to test all the available automated malware analysis tools freely accessible on the web that use a sandbox to analyze the file, and also all the main debuggers for Windows operating system, since many of these techniques are intended as anti-debugging tricks, to analyze and understand what kind of countermeasures, the developers of these tools, use against evasive malware and their evasion techniques. The results we collected showed us that sandbox developers made many efforts to hide the presence of a virtualized environment, indeed almost all the evasion techniques based on looking for the existence of certain files, processes, drivers, file system structure, Windows ID, username, registry, etc. that could reveal the presence of a sandbox failed. While, always from the obtained results, we noticed that other approaches to do evasion like detecting time discrepancies during the execution of certain instructions or delay the execution of the malware payload of a certain amount of time, in such a way that the time dedicated for the analysis by the sandbox is not enough to detect the malicious activities, are still very powerful methods. Sandbox developers introduced some countermeasures against these families of evasion techniques, but they are still not enough because, for example, they focus only on the most famous functions used to stall the execution, like the *Sleep* API, while we saw that malware authors always come up with new methods and tricks. With this work we also tried to show the obtained results quickly and intuitively, so that we can provide to

everyone an evaluation of the different automated malware analysis systems available at the moment, underlining their strengths and weaknesses concerning the problem of evasive malware, doing so we also provided a classification of the different evasion techniques showing which are the more powerful and useful to do evasion. Another goal of our work was to test also the resistance of the main Windows debuggers against evasive malware because they are fundamental tools for dynamic malware analysis, often used by security specialists, also in this case thanks to the tool we developed we wrapped around a malware sample different evasion techniques intended precisely against debuggers, and we saw that almost all of these anti-debugging tricks work fine usually making crash the debugger or stopping the execution on certain instructions. Sincerely, above all from the still updated projects, we expected a greater resistance, but this is not a big issue, unlike in the case of automated tools, because in the end, the crash of the debugger during the execution of a sample is a big suspicious alert for the analysts.
In conclusion this work provides to everyone a clear assessment of the available automated malware analysis tools freely accessible, underlining that this study does not consider the premium versions that could obtain better results, and it provides also a useful tool to test the resistance of sandbox environments and debuggers, that collects a number of evasion techniques greater than previous works, but above all, it allows to apply them to any executable.

This work has been conducted with the hope of contributing to the study against evasive malware.

BIBLIOGRAPHY

[1] F. Al Ameiri and K. Salah. "Evaluation of popular application sandboxing." In: *2011 International Conference for Internet Technology and Secured Transactions*. 2011, pp. 358–362.

[2] Dr Al-Anezi. "Generic Packing Detection Using Several Complexity Analysis for Accurate Malware Detection." In: *International Journal of Advanced Computer Science and Applications* 5 (Jan. 2014). DOI: 10.14569/IJACSA.2014.050102.

[3] Aldeid. *X86-assembly/Instructions*. URL: https://www.aldeid.com/wiki/X86-assembly/Instructions.

[4] Yaniv Assor. *Anti-VM and Anti-Sandbox Explained*. URL: https://www.cyberbit.com/blog/endpoint-security/anti-vm-and-anti-sandbox-explained/.

[5] *Cuckoo Sandbox - Automated Malware Analysis*. URL: https://cuckoosandbox.org/.

[6] *Cybercrime Damages $6 Trillion By 2021*. URL: https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016.

[7] D12dox34X. *Anti-Debugging and Anti-VM Techniques and Anti-Emulation*. URL: https://resources.infosecinstitute.com/topic/anti-debugging-and-anti-vm-techniques-and-anti-emulation/.

[8] Zero Day. *Anti Reversing Techniques*. URL: https://zer0day.tistory.com/335.

[9] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. "A Survey on Automated Dynamic Malware-Analysis Techniques and Tools." In: *ACM Comput. Surv.* 44.2 (Mar. 2008). ISSN: 0360-0300. DOI: 10.1145/2089125.2089126. URL: https://doi.org/10.1145/2089125.2089126.

[10] Peter Ferrie. *The "Ultimate" Anti-Debugging Reference*. URL: https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf.

[11] *Google Safe Browsing service*. URL: https://transparencyreport.google.com/safe-browsing/overview?hl=en.

[12] Bojan Jovanović. *Malware statistics – You'd better get your computer vaccinated*. URL: https://dataprot.net/statistics/malware-statistics/.

[13] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. "The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code." In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. CCS '11. Chicago, Illinois, USA: Association for Computing Machinery, 2011, 285–296. ISBN: 9781450309486. DOI: 10.1145/2046707.2046740. URL: https://doi.org/10.1145/2046707.2046740.

[14] Oleg Kulchytskyy and Anton Kukoba. *Anti Debugging Protection Techniques with Examples*. URL: https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software.

[15] Check Point Software Technologies LTD. *Anti-Debug Tricks*. URL: https://anti-debug.checkpoint.com/.

[16] Malwarebytes Lab. *2020 State of Malware Report*. URL: https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report.pdf.

[17] Minerva Labs. *Evasive Malware: Learning by Example*. URL: https://blog.minerva-labs.com/evasive-malware-learning-by-example.

[18] Christian Stigen Larsen. *Embedding binary data in executables*. URL: https://csl.name/post/embedding-binary-data/.

[19] John Leitch. *Anti-Debugging With Exceptions*. URL: http://www.autosectools.com/anti-debugging-with-exceptions.pdf.

[20] LordNoteworthy. *Al-Khaser v0.80*. URL: https://github.com/LordNoteworthy/al-khaser.

[21] Net MarketShare. *Operating System Share by Version*. URL: https://netmarketshare.com/operating-system-market-share.aspx.

[22] *Mingw-w64*. URL: http://mingw-w64.org/doku.php.

[23] N. Miramirkhani, M. P. Appini, N. Nikiforakis, and M. Polychronakis. "Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts." In: *2017 IEEE Symposium on Security and Privacy (SP)*. 2017, pp. 1009–1024. DOI: 10.1109/SP.2017.42.

[24] Dr. Farrukh Shahzad Muhammad Hasib Latif. *Increased Use of WMI for Environment Detection and Evasion*. URL: https://www.fireeye.com/blog/threat-research/2016/10/increased_use_ofwmi.html.

[25] Anitta Patience Namanya, Jules Pagna Diss, and Irfan Awan. "Evaluation of automated static analysis tools for malware detection in Portable Executable files." In: Sept. 2015.

[26] Tomasz Nowak. *NTAPI Undocumented Functions*. URL: http://undocumented.ntinternals.net/.

[27] Yoshihiro Oyama. "Investigation of the Diverse Sleep Behavior of Malware." In: *Journal of Information Processing* 26 (2018), pp. 461–476. DOI: 10.2197/ipsjjip.26.461.

[28] PurpleSec. *2020 Cyber Security Statistics*. URL: https://purplesec.us/resources/cyber-security-statistics.

[29] *Selenium Client Drive*. URL: https://www.selenium.dev/selenium/docs/api/py/index.html.

[30] Rob Sobers. *110 Must-Know Cybersecurity Statistics for 2020*. URL: https://www.varonis.com/blog/cybersecurity-statistics.

[31] Joshua Tully. *An Anti-Reverse Engineering Guide*. URL: https://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide.

[32] *VIrtual Machine Detection Techniques*. URL: https://shasaurabh.blogspot.com/2017/07/virtual-machine-detection-techniques.html.

[33] Shaul Vilkomir-Preisman. *Anti-Virtualization Malware*. URL: https://www.deepinstinct.com/2019/10/29/malware-evasion-techniques-part-2-anti-vm-blog/.

[34] *VirusTotal*. URL: https://www.virustotal.com/.

[35] Microsoft Windows. *Win32 API doc*. URL: https://docs.microsoft.com/en-us/windows/win32/api/.

[36] *Yara - The pattern matching swiss knife for malware researchers (and everyone else)*. URL: https://virustotal.github.io/yara/.

[37] aortega. *Pafish: Paranoid Fish*. URL: https://github.com/a0rtega/pafish.

[38] ntopcode. *Anatomy of the Process Environment Block (PEB) (Windows Internals)*. URL: https://ntopcode.wordpress.com/2018/02/26/anatomy-of-the-process-environment-block-peb-windows-internals/.

[39] ytisf. *theZoo - A Live Malware Repository*. URL: https://github.com/ytisf/theZoo.

[40] zwclose7. *The KUSER_SHARED_DATA structure*. URL: http://www.rohitab.com/discuss/topic/42325-the-kuser-shared-data-structure/.