

Master's Thesis

Bypassing modern sandbox technologies

An experiment on sandbox evasion techniques

Gustav Lundsgård
Victor Nedström



Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, 2016.

Bypassing modern sandbox technologies

An experiment on sandbox evasion techniques

Gustav Lundsgård
Victor Nedström

Department of Electrical and Information Technology
Lund University

Advisor: Paul Stankovski

June 21, 2016

Abstract

Malware (malicious software) is becoming an increasing problem, as it continuously grows both in numbers and complexity. Traditional, signature based anti-virus systems are often incapable of detecting new, sophisticated malware, which calls for more advanced tools. So called *sandboxes* are tools which automate the process of analyzing malware by actually running them in isolated environments and observing their behavior. Although this approach works very well in theory, some malware have recently begun deploying *sandbox detection techniques*. With the help of these techniques, malware may detect when they are being analyzed and manage to evade the sandbox by hiding their malicious behavior.

The authors of this Master's Thesis have developed and compared different types of sandbox detection techniques on five market leading products. It was shown that an average of roughly 43% of the detection techniques developed were capable of both detecting and bypassing the sandboxes, and that the best performing sandbox caught as much as 40% more of the techniques than the worst. Patterns of weaknesses were noticed in the sandboxes, affecting primarily the limited hardware and lack of user interaction - both of which are typical sandbox characteristics. Surprisingly, the time for which the sandbox vendors had been developing their sandboxing technology seemed to have no positive impact on the result of their product, but rather the other way around. Furthermore, some detection techniques proved very efficient while being trivial to develop. The test results have been communicated to the sandbox vendors, and the authors are of the belief that the sandboxes could be quite significantly improved with these results as a guideline.

Acknowledgements

This Master's Thesis is a result not only of our own work but also of the efforts of people we have had the privilege to work with. Without you, this Master's Thesis would not be what it is today.

First and foremost, we would like to warmly thank our supervisor at the department of Electrical and Information Technology, Paul Stankovski, for his commitment and care throughout the entire process.

Secondly, we want to express our sincere gratitude towards Coresec Systems AB for giving us the opportunity to write our Master's Thesis on a topic close to our hearts. We would especially like to thank our two supervisors, Stefan Lager and David Olander, for their continuous support, technical advisory and encouragement. Furthermore, both Philip Jönsson and Holger Yström deserve many thanks for their efforts in helping and sharing their technical expertise. We are very grateful to all of you for everything we have learnt during the process and are looking forward to working with you in the future.

Lastly, we would like to thank The Swedish National Board of Student Aid, Centrala Studiestödsnämnden (CSN), for their investment in our studies.

Acronyms

A

API - Application Programming Interface

C

C&C - Command and Control (server)

CPU - Central Processing Unit

D

DLL - Dynamic Link Library

DMZ - Demilitarized Zone

DNS - Domain Name Server

G

GPU - Graphics Processing Unit

GUI - Graphical User Interface

H

HTTP - Hypertext Transfer Protocol

I

IAT - Import Address Table

ICMP - Internet Control Message Protocol

IRC - Internet Relay Chat

M

MAC - Media Access Control

P

PE - Portable Executable

R

RDP - Remote Desktop Protocol

V

VLAN - Virtual Local Area Network

VM - Virtual Machine

VMM - Virtual Machine Monitor

VPN - Virtual Private Network

Glossary

C

Command and Control (C&C) server: a server to which malware, after infecting a target host, connects to download files or receive additional instructions.

D

Debugger: enables a program to be examined during execution. Most debuggers can run programs step by step.

F

False positive: a benign file causing a "false alarm" in an anti-malware system, commonly because its behavior resembles that of malicious files.

G

Golden image: an operating system image configured according to best effort.

H

Hypervisor: software that enables virtualization by creating and running virtual machines.

M

Malware sample: a single copy of malware being subject to analysis.

R

Ransomware: a type of malware that encrypts files on a file system and demands a ransom to be paid in exchange for the decryption key.

S

Snapshot: a preserved state in a virtual machine.

Z

Zero-day threat: a previously unseen threat.

Table of Contents

1	Introduction	1
1.1	Background	1
1.2	Problem Description	1
1.3	Purpose	2
1.4	Method	2
1.5	Scope	3
1.6	Report Layout	4
2	Theory	5
2.1	Malware	5
2.1.1	Origin and purpose	5
2.1.2	Types of malware	6
2.2	Malware analysis	7
2.2.1	Static Analysis	7
2.2.2	Dynamic Analysis	8
2.3	Malware analysis detection	8
2.4	The Portable Executable file format	9
2.5	Running executable files	11
2.6	Virtual machines	11
2.7	Sandboxes	12
2.7.1	Detection of virtual machines and sandboxes	13
2.8	DLL injection and API hooking	14
2.9	Sandbox analysis procedure	15
2.10	Related work	17
3	System Description	19
3.1	Introduction	19
3.2	Vendors and sandboxes	19
3.2.1	Vendor A	19
3.2.2	Vendor B	20
3.2.3	Vendor C	21
3.2.4	Vendor D	22
3.2.5	Vendor E	22
3.2.6	Sandbox specification summary	22

4	Experiments	25
4.1	Introduction	25
4.2	Test case design	25
4.3	Test method, environment and configuration	27
4.3.1	Inline deployment	29
4.3.2	Upload deployment	30
4.4	Test cases	31
4.4.1	Timing	32
4.4.2	Process	33
4.4.3	File	33
4.4.4	Environment	34
4.4.5	Hardware	35
4.4.6	Network	35
4.4.7	Interactive	36
4.5	Test criteria	37
4.6	Choice of malware	37
5	Results	39
5.1	Introduction	39
5.2	Results per category	39
5.2.1	Timing	40
5.2.2	Process	42
5.2.3	File	44
5.2.4	Environment	46
5.2.5	Hardware	48
5.2.6	Network	50
5.2.7	Interactive	52
5.3	Result of golden image configuration	54
5.4	Result summary	55
6	Discussion	57
6.1	Sandbox technologies and result patterns	57
6.2	Weakness patterns based on detection technique	60
6.3	Simplicity contra efficiency in detection techniques	61
6.4	System lifetime in relation to test results	62
6.5	General discussion and sources of error	63
7	Conclusions	67
7.1	Future work	68
	References	69
A	Test case simplicity-efficiency score	73

List of Figures

2.1	Flow chart of an inline API hook.	14
2.2	Communication between user mode and kernel mode components. . .	15
3.1	Relative vendor size measured in number of employees	20
3.2	Relative vendor age	21
4.1	Test case components overview	26
4.2	Test case flow chart	26
4.3	A simplified overview of the test environment.	28
4.4	Detailed view of the test environment including network interfaces. .	30
4.5	A detailed view of how the malware is hidden as a resource in the test case [1].	32
5.1	Timing - detected malware per vendor.	41
5.2	Process - detected malware per vendor.	43
5.3	Files - detected malware per vendor.	45
5.4	Environment - detected malware per vendor.	47
5.5	Hardware - detected malware per vendor.	49
5.6	Network - detected malware per vendor.	51
5.7	Interactive category - detected malware per vendor.	53
5.8	Malware detection rate per vendor.	55
5.9	Average malware detection rate per category and coefficient of variation. .	56
6.1	Sandbox detection rate per vendor where the test cases detected the sandbox	59
6.2	Simplicity contra efficiency score per category.	62

List of Tables

3.1	Sandbox specifications and features	23
5.1	Timing results statistics.	40
5.2	Timing category results per vendor.	40
5.3	Process results statistics.	42
5.4	Process category results per vendor.	42
5.5	Files results statistics.	44
5.6	File category results per vendor.	44
5.7	Environment results statistics.	46
5.8	Environment category result per vendor.	46
5.9	Hardware result statistics.	48
5.10	Hardware category result per vendor.	48
5.11	Network result statistics.	50
5.12	Network category results per vendor.	50
5.13	Interactive result statistics.	52
5.14	Interactive category results per vendor.	52
5.15	Result between golden and original images.	54
5.16	Overall result statistics	55
A.1	Timing category	73
A.2	Process category	73
A.3	Files category	74
A.4	Environment category	74
A.5	Hardware category	75
A.6	Network category	75
A.7	Interactive category	76

1.1 Background

Coresec Systems ("Coresec"), is a European cyber security and networking solutions company represented in Sweden, Denmark, Norway and the Netherlands. The headquarters are located in Malmö, where around 100 of Coresec's 250 employees are stationed. Coresec consists of several departments, the *Security Operations Center* being one of them. The Security Operations Center provides 24/7 coverage and consists of security analysts who are primarily concerned with handling security issues or potential issues of Coresec's customers. On a daily basis, these analysts face the task of dealing with and analyzing malware.

Malware is a general term used to describe malicious computer software. Both malware and cyber security as a whole are rapidly growing and changing areas, and leading cyber security companies publish annual reports indicating current trends [2]. According to statistics focusing on "E-crime and malware" in particular, almost 320 million new malware variants were detected only in 2014, which was a 25% increase compared to 2013 [3]. Furthermore, the number of *ransomware* - a certain type of malware - more than doubled, and a total of 28% of all malware was *virtual machine aware* according to the same source.

1.2 Problem Description

Due to the sheer amount of both new and existing malware that appear on the web and the fact that analyzing malware is (very) time consuming, it is infeasible for malware analysts to manually analyze all suspicious files to conclude whether they may or may not be malicious [4]. Instead, analysts must rely on automatic anti-malware tools to take care of the majority of these suspicious files, and carefully investigate only a fraction of these.

Fighting malware is a difficult task as malware continuously becomes more and more sophisticated. The situation is an arms race between malware developers and malware analysts, where malware developers keep developing methods to obstruct analysis techniques deployed by their counterpart. Traditional automated anti-malware solutions, such as anti-virus software, used to focus mainly on studying so called *file signatures*, which are "identifiable pieces of known ma-

licious code" [5] [6]. However, this type of analysis turned out less and less successful over time, as the usage of different obfuscation techniques grew more popular among malware developers. For instance, by encrypting pieces of malware or simply creating different variants, malware could stay under the radar. As a consequence, instead of only studying signatures, some anti-malware software began to implement functionality to be able to execute the suspicious files in isolated environments and study their behavior. By closely monitoring processes and changes to the file system on the machine in the isolated environment, the tools could determine whether or not the behavior of the suspicious files were indeed malicious. However, as malware developers became aware of this, they started developing so called virtual machine or sandbox aware malware: malware that deploy techniques to determine whether they are being executed in an isolated environment or on a real host. Detecting an isolated environment would result in the malware not disclosing its malicious behavior, as it knows it is probably being closely monitored.

1.3 Purpose

Vendors of dynamic analysis systems claim to have good detection rates and low amounts of false positives for obvious reasons. However, exactly how their systems work is rarely disclosed, as this information could potentially be exploited by malware developers. On the contrary, this secrecy also makes it hard for end-users of the systems to actually know how well the products manage to identify and hinder threats. The purpose of this Master's Thesis was to develop test cases to examine top-of-the-line dynamic analysis systems and to evaluate how simple it is for malware to avoid detection using different detection techniques.

The aims were to answer the following questions:

- Are there any patterns in the test results based on the type of technology used in the different systems?
- Are there any patterns of weaknesses in the systems based on the type of detection technique?
- How is simplicity weighted against efficiency in the detection techniques?
- Is any type of technique significantly more efficient than the others?
- Does the lifetime of a system have a significant impact on its ability to withstand the detection techniques?

1.4 Method

As a first step, a comprehensive literature study and information research was performed to gain necessary background knowledge on the area as a whole; the PE file format, sandboxes and virtual machine theory and various malware aspects are three of the areas that were studied. Thereafter, more in-depth research

targeting existing sandbox detection techniques was made. These techniques were divided into categories to form a base to develop test cases on.

A suitable malware to use in the tests was chosen based on recommendations from an experienced malware analyst on Coresec. Verification was made that this malware was indeed recognized as malicious by all sandboxes to be tested. Furthermore, a decision on how to bundle the malware with a detection technique, to form a test case, was made based on information research and practical experiments.

The next big step in the process was implementing the test cases, each based on one detection technique. New detection techniques were added to the already existing ones as new ideas emerged during development. All test cases were tested, trimmed and tuned on local machines in order to ensure their functionality prior to the actual test phase. Some of the test cases were also run on an open source sandbox (not to be included in the tests) [7].

Prior to running the tests, the test environment was designed and set up. First, a number of *Virtual Local Area Networks* (VLAN) were created, each with a certain "trust level" depending on what type of traffic would pass through the network. Second, two ESXi hypervisor servers were installed and a number of virtual machines on them in turn, including virtual sandboxes, a web server and two Microsoft Windows ("Windows") clients to be used in the experiments. Thereafter the physical sandbox appliances were installed in the lab, and finally all sandboxes both virtual and physical were configured and prepared for testing.

The last step in the experiment process was running all test cases on each sandbox and carefully documenting the results. The results were analyzed and conclusions drawn to answer the project aims.

1.5 Scope

Because of the limited time frame and resources dedicated to the project, some limitations had to be made to the scope accordingly. This mainly affected the detection techniques developed, which only focus on evading sandbox based solutions, i.e. automated tools performing dynamic analysis. These tools typically reside in gateways and not in endpoints, i.e. on dedicated machines in networks rather than hosts. In a real scenario, where an executable like the one used in project was to be delivered to a target host, dynamic analysis tools would typically be complemented by automated, static analysis tools such as firewalls and antivirus software. In such a scenario, the malware would preferably have to contain some self modifying code or similar to avoid signature detection. In this Master's Thesis, only the effort required was made in order not to get considered malicious in the static analysis.

Furthermore, in a real scenario, it would be of great interest what activities the actual malware performed on the host after successful delivery. Malware would typically attempt to communicate to a (C&C) server using the network of the infected host, to download additional files or instructions. However, from the perspective of this project, this is completely ignored.

1.6 Report Layout

This report is organized as follows:

- **Chapter 1** contains an introduction, briefly presenting some background information about malware in general. The problem area is then presented, which is followed by the goals, the selected method to satisfy these goals and the scope of the project.
- **Chapter 2** presents relevant theory to the reader, which includes malware and malware analysis, the PE file format, executable files in Windows, virtual machines and sandboxes, sandbox detection, API hooking and lastly the sandbox analysis procedure. The related work in the area is also presented.
- in **Chapter 3** a system description is given, where the sandbox vendors are individually (but anonymously) presented together with their respective sandbox.
- **Chapter 4** presents all practical details related to the experiments except for the outcome. This includes information about e.g. the test environment, the malware used, the different techniques for sandbox detection, the test cases and how the tests were executed.
- in **Chapter 5** the test results of the tests are given as tables and diagrams.
- **Chapter 6** contains a discussion about the results, both in the light of the project aims and on a more general level. Furthermore, some potential sources of error are discussed.
- lastly, in **Chapter 7**, conclusions related to the previous discussion and project aims are drawn based on the test results. Additionally, some suggestions for future work are given.

2.1 Malware

Malware is an abbreviation for *malicious software* and there are numerous definitions of what exactly malware is: some sources describe malware simply as software which "runs much like other software" [8], while others include the effects of malware in the definition and describe it as "any software that [...] causes harm to a user, computer or network" [5]. Some even include details about the behavior in the definition [9]. Examples of such behavior could be "stealing user data, replicating, disabling certain security features, serving as a backdoor, or executing commands not intended by the user" [8]. Malware typically only satisfy one or a few of these "requirements", and are commonly divided into categories based on their type of behavior.

2.1.1 Origin and purpose

There are different categories of people who develop malware, each having their own methods and reasons for doing so. These categories include everything from people who do it out of curiosity without evil intentions to those for whom developing malware is serious business and something to make a living on [10]. While the former used to be the more commonly seen one, today the situation is the opposite and the majority of malware developers have criminal purposes.

There are huge sums up for grabs for cybercriminals capable of developing potent malware. For instance, they could target the systems of a specific organisation or company and develop malware which aim to extract secret information, which could then be sold to competitors or used for blackmailing [11]. Furthermore, malware developers could target random systems and attempt to gain control over as many of them as possible, and from there on take a number of various approaches depending on what type of malware they prefer. The number of infected hosts are often critical to a malware developer: the more systems he or she is able to infect, the more money he or she can make [12]. Since Windows is extremely dominant on the operating system market today (having more than 90% of the market shares) an overwhelming majority of all existing malware target Windows x86 operating systems (since x86 executables run on x64 systems but not vice versa) to maximize the probability of infecting a random system (where

the operating system is unknown to the malware developer) [13].

2.1.2 Types of malware

The facts that widely used malware categories partly overlap each other and that malware often span multiple categories means that there exists no single, correct categorization of malware. The following categorization will be used as a reference in this Master's Thesis [5]:

- **Backdoors** allow attackers to easily get access to infected systems by opening up a way to connect while avoiding authentication. Backdoors grant attackers potentially "full control of a victim's system without his/her consent" [14]. Once connected, the attacker can execute commands on the system to e.g. install additional malware or attack other hosts on the network.
- **Botnets** resemble backdoors, but the infected hosts are not controlled individually but collectively using an *Internet Relay Chat* (IRC) channel or similar for instructions [15]. Instead of a single system being of interest to the attacker, it is the amount of infected hosts that is of value. Hosts that have become part of botnets - sometimes called zombies - are commonly used in *Distributed Denial of Service* (DDoS) attacks or cryptocurrency mining, where an attacker relies on a big number of hosts to send big amounts of data or perform resource consuming calculations.
- **Downloaders** are used to download additional malware, as indicated by the name. When successfully infecting a system, a downloader fetches malware of other types to exploit the infected system [16].
- **Information-stealing malware**, also referred to as **spyware** or simply **stealers**, come in a number of different flavors. Their common feature is that they aim to steal user account credentials for e.g. e-mail or bank accounts, which are sent to the attacker. *Keyloggers* and *sniffers* are two examples of such malware which log keyboard input and network traffic respectively.
- **Launchers** are simply used, as the name implies, to launch other malware. Although this could seem pointless, something launchers could potentially achieve is stealth, which will be utilized in this Master's Thesis.
- **Ransomware** or **lockers**, which have increased in popularity lately, encrypt parts of a system and demand a ransom to be paid in exchange for the decryption key [3]. Ransomware is especially problematic since nothing but the decryption key will be able to recover the encrypted data (assuming brute-force is infeasible).
- **Rootkits** are malware that focus on hiding not only their own existence and behavior but also that of other, bundled malware such as backdoors. This is done using "stealth techniques which [...] prevent itself from being discovered by system administrators" [17].
- **Scareware** disrupt the user of an infected system by continuously notifying him or her that the system has been infected by malware. However,

scareware also offers to remove themselves from the infected system in exchange for a fee.

- **Spam-sending malware** use infected machines to spread spam to others, typically via e-mail.
- **Worms** or **viruses** are malware that focus on spreading and infecting other systems. Worms or viruses could be spread using e.g. local networks or physical storage media.

2.2 Malware analysis

The ultimate goal of malware analysis is if to conclude whether or not a file is malicious. In addition to this, it is often of interest to gain more information about the file and aim to answer the following three questions [5]:

- How does the malware work, i.e. what does it do and how?
- How can the malware be identified to avoid future infections?
- How can the malware be eliminated?

Prior to these three steps of the analysis process, it is assumed to have been confirmed that the malware being subject to analysis is indeed malware and not a *false positive*. False positives are "normal data being falsely judged as alerts", which in this case means benign files whose behavior might be similar to that of malicious files [18].

Two basic approaches exist for analyzing malware: static and dynamic analysis [5]. Both approaches apply not only to automatic but also to manual analysis. Typically, when analyzing malware, a quick static analysis is performed initially which is followed by a more thorough, dynamic analysis [5].

2.2.1 Static Analysis

During static analysis, malware samples are examined without being executed. Essentially this means that an analyst or a tool attempts to extract as much information as possible about a file to conclude whether or not it is malicious. Automated static analysis, carried out by anti-malware software, focus mainly on the signatures of files, i.e. on finding presence of malicious code that is present in databases containing known malicious data [19]. Both automated and manual static analysis could for instance also include studying strings in the source code (searching for e.g. URL:s or suspicious names), exported and imported functions or resources used by the file to draw conclusions about its behavior. Manual static analysis is significantly more time consuming than automated analysis, but can be done more carefully with greater attention to details [5].

Whether or not a static analysis turns out successful depends to a large extent on if the malware is encrypted or not. Many malware developers nowadays obfuscate code by encrypting ("packing") it, which means that little information can be extracted from encrypted files compared to those in plain text [1]. Encrypting

the code also has the advantage - from a malware developer's perspective - that signatures for which there would be a match if the malware was not encrypted may be avoided [20].

Static analysis is quick and straightforward, but unfortunately has some inevitable drawbacks [5]. First of all, the instructions of the malicious file are ignored, since they are never loaded into memory (this happens first when the file is executed, see section 2.5). Second, static analysis struggles with packed files. Furthermore, since the malware is never executed, its actual effects on the system cannot be observed. As a consequence, static analysis alone is insufficient most of the time and has proven ineffective to fight sophisticated malware [21].

2.2.2 Dynamic Analysis

In a dynamic analysis the purpose is to execute the malware and observe its behavior and effects on the system during runtime. Dynamic analysis, just like static analysis, can be performed both automatically and manually. When done automatically, the file system, processes and registry of the operating system are being closely monitored during the execution of the malware, and any malicious operations that are detected may indicate that the file being executed is malicious [5]. Manual dynamic analysis typically involves *debugging*, which is "the process of identifying the root cause of an error in a computer program" by systematically stepping through the program [22].

Dynamic analysis has the possibility of revealing lots of information that static analysis simply is not capable of. First of all, dynamic analysis can observe the behavior of malware during runtime, which static analysis can not. Furthermore, since malware that is encrypted or packed decrypts itself when executed, the malware will be laying in plain text in the memory during most of the execution [5]. This means that a malware analyst - by debugging the malware properly - will be able to dump a plain text version of the malware to disk to e.g. make a static analysis. Perhaps most importantly though, dynamic analysis is - in contrast to static analysis - capable of detecting so called "zero-day threats", which are previously unseen malware for which no signatures exist [23].

A dynamic analysis takes more time than a static analysis to perform, and a thorough, manual dynamic analysis is also far more difficult than a manual static one. However, due to its obvious advantages, dynamic analysis has grown more and more important over the years to cope with the latest and most advanced malware threats.

2.3 Malware analysis detection

It has already been mentioned that malware is commonly packed to obstruct the task of malware analysis. However, this obstructs mainly static analysis; dynamic analysis could still, with some effort, get around packed malware. What countermeasure could malware use to obstruct dynamic analysis as well?

A thorough, manual dynamic analysis is hard to counter; skilled malware analysts will likely find ways around even the most ambitious obstruction attempts

eventually. Normally though, a manual analysis takes place first when an automated anti-malware solution has indicated a need for this on a certain file that seems suspicious. Malware that manage to evade automated anti-malware tools may stay under the radar for quite a long time, and may not be detected until a host starts showing signs of infection. Furthermore, well thought-out obstruction techniques, such as code obfuscation, might fool less experienced malware analysts, who fail to successfully debug malware and mistakenly takes them for being harmless. At the very least, the process of analysing malware will be slowed down if the malware developers have utilized obstruction techniques.

Since malware is never executed during static analysis, this type of analysis cannot be detected by the malware themselves. On the contrary, since malware is being executed during dynamic analysis, it has the opportunity to detect the ongoing analysis and take actions accordingly [24]. For instance, the malware could investigate the environment it is being executed in by looking at hardware or running processes, and if it matches certain criteria, the malware could assume that it is being analyzed and choose not to do anything malicious. If no signs of an ongoing analysis are found, the malware will instead assume that it is being executed on an unsuspecting host and execute as intended.

2.4 The Portable Executable file format

The Portable Executable (PE) file format is used in modern versions (x86 and x64) of Windows operating systems. It defines the structure for a set of different file types, among which executables and *Dynamic Link Libraries* (DLL) are two of the most commonly seen. Files that follow this structure can be loaded and executed by the *Program loader* (also called *PE loader*) in Windows.

Executable files can be dissected into two high-level components: a header part containing meta-data, and a sections part containing the actual data [25]. In order to avoid confusion - since naming conventions regarding the PE file format are somewhat inconsistent - these high-level components will be referred to as *executable's header* and *executable's sections* respectively in this Master's Thesis.

The executable's header, in turn, consists of the following five sections:

- the **DOS header**, which is static and informs the operating system that the file cannot be run in DOS environments. The DOS header always begins with two bytes that are equal to "MZ", which is followed by the *DOS stub*. The DOS stub contains bytes that translate to "This program cannot be run in DOS mode". The fact that these two signatures are static makes the DOS header very easy to identify when inspecting the binaries of an executable file.
- the **PE header**, sometimes referred to as **NT Header** or **COFF Header**, which also begins with a static signature, "PE". Furthermore, it contains information about which processor architecture the program will run on, the number of sections in the executable (see *Sections table* below), a file creation timestamp and a few other fields indicating sizes and addresses of other fields in the file. Inconsistencies between these fields and the actual values they refer to are often detected and flagged as malicious by

anti-malware software, since these inconsistencies could typically be introduced by someone trying to tamper with the file.

- the **Optional header**, which contains general information about the file and states for instance whether it is a 32 or 64-bit binary, which version of Windows is required to run the file and the amount of memory required. More importantly, like the PE header, the Optional header also contains lots of size and pointer fields related to data in the file, such as where execution starts (called *Address of Entry Point*) and where different parts of the file should be placed in memory when loaded by the Windows program loader. In other words, the information in the Optional header is essential for Windows to be able to run the file, and as with the PE header, there are numerous fields where an incorrect value will cause the program to crash and/or raise flags in anti-malware software analyzing the file.
- the **Data directories**, which contain addresses to different parts of the data in the executable's sections that primarily concern imports of data from external libraries (typically DLL:s) but also exported functions to be used by other executables. The *Import Address Table* (IAT), which contains addresses of DLL:s to load, is important in this context and will be further explained in section 2.8.
- the **Sections table**, which describes the executable's sections and defines how they are loaded into memory. For each section listed in the sections table, a pointer to the corresponding data in the executable's sections is defined together with the size of that data block, to inform the Windows program loader where in the memory to load that section and exactly how much memory needs to be allocated for it. The sections table is of special interest in this Master's Thesis, and therefore deserves to be described in slightly greater detail than the other parts of the executable's header. In general, the Windows program loader does not bother about the contents of the different sections; it simply loads them into the specified location in memory. However, there are 24 section names which are reserved for sections with special purposes [26]. Out of these reserved names, a handful are common and exist in most applications. Since the section names are decided by the compiler they may differ slightly, but typically the following naming conventions are used:
 - the **.text** section, sometimes referred to as **.code**, references the executable program code.
 - the **.rdata** section references read-only, compiler generated meta-data such as debugging information.
 - the **.data** section references static source code data such as strings.
 - the **.reloc** section references data used by the Windows program loader to be able to relocate executables in memory.
 - the **.rsrc** section references resources used by the executable. Resources must be of certain types; examples of commonly seen resources are icons and version information, but it could also be components from

other executables such as dialog boxes [27]. One specific type allows for "application-defined resources", i.e. raw data [27]. As will be demonstrated later in this Master's Thesis, this resource type could be used for instance to include an executable file as a resource within another executable file.

2.5 Running executable files

To build an executable file, the source code must be *compiled* by a compiler. Simply put, a compiler transforms the source code into machine code that the operating system can understand. During compilation, the compiler is also responsible for linking the contents of other files, such as functions available in external libraries, to the executable which it may be dependent of to be able to run [28]. This linking can be either *static* or *dynamic*, which means that library functions will be either directly put into the executable - i.e. as a part of the executable itself - or linked to the executable via a DLL on the operating system respectively. DLL files are essentially libraries of functions, which can be used by other programs [29]. One such example is the Windows Application Programming Interface (API), which is made up of a set of DLL:s.

When running an executable file in Windows, initially the Windows program loader loads the program into memory. When this process is finished, the operating system can start parsing the PE header. When parsing the PE header, the operating system identifies all imported DLL:s required by the program and loads them into memory, so that the executable can use them during execution.

After the DLL:s are loaded into memory, the program loader finds the source code at the *entry point*. The entry point is the relative address of the "starting point" of the source code, i.e. where the execution of the program starts [26]. During executing, each line of code - which has been translated to machine code - is executed and the system performs the instructions of the program.

2.6 Virtual machines

The concept of virtualization is to "map the interface and visible resources [of a system] onto the interface and resources of an underlying, possibly different, real system" [30]. In other words, virtualizing a system means to create a virtual version of it which does not run directly on hardware but on the software of another system. This virtual version is called a *Virtual Machine* (VM) and is commonly divided into two types: process virtual machines and system virtual machines [30]. The former is used to create an environment to run certain programs, e.g the Java Virtual Machine, while the latter aims to imitate an entire operating system. The system that runs inside the virtual machine is called a guest and the platform where the virtual machine runs is called a host. The software on the host that enables the actual virtualization is called a *hypervisor* or *Virtual Machine Monitor* (VMM).

There are several benefits of using virtual machines in different scenarios.

One issue it solves is that of having dependencies on defined interfaces of different operating systems, making it possible to have several different systems on the same hardware. This is used in e.g. cloud computing, where one powerful server hosts several virtual servers which share hardware, ultimately reducing both hardware and energy costs [31]. Virtual machines and their hypervisors also enable the use of snapshots. A snapshot is used to "preserve the state and data of a virtual machine at the time you take the snapshot", enabling the possibility of going back to an earlier state of the system [32]. This functionality has many upsides and is fundamental for testing malware, as being able to revert potential changes made to the system by the malware is necessary.

Virtual machines also provide the vital ability to control malware and their environment more conveniently than a physical computer. A guest can be totally isolated from the host and also runs with reduced privileges in comparison, allowing the hypervisor to monitor or even intercept actions and events taking place on the guest. The guest system is also unaware of the fact that it resides inside a virtual environment and not directly on hardware, since the hypervisor virtualizes the hardware and fools the guest into believing that it has a machine for itself and direct access to the hardware [30].

When speaking about virtual machines, a common term is *image* or *system image*. An image is very similar to a snapshot and the terms are sometimes used interchangeably, but images do not preserve the exact state of a (running) machine; instead, images have to be booted.

2.7 Sandboxes

In the context of computer security, the term *sandbox* or *sandboxing* is used to describe systems which utilize virtualization technology to realize the isolation of a host. The purpose of a sandbox is to provide an isolated and monitored environment for a program to be executed in, to be able to distinguish whether or not the program is malicious.

Sandboxes may operate in slightly different manners and can be either system virtual machines (see section 2.6) or programs which enable other executables to be run in a sandboxed mode, such as Sandboxie [33]. In the context of dynamic malware analysis, sandboxes utilize system virtual machines since this gives the malware sample being executed the impression of being on a "real" system although it is completely isolated. Running active malware can have severe consequences if not controlled properly, as they could e.g. spread to other hosts on the same network. Sandboxes are typically deployed somewhere in the network whose endpoints it aims to protect. These endpoints could be clients, servers or other resources on a local network, which are referred to as *hosts* hereinafter in this Master's Thesis. The purpose of a sandbox is to analyze files bound for hosts on the network, to conclude whether the files are malicious or not. Malicious files are blocked, while benign file are passed on to the hosts.

System virtual machine sandboxes can use either images or snapshots when starting a virtual machine. Both images and snapshots can be configured according to preferences, and furthermore, snapshots can be in any desired state. Either

way, it is desirable that the virtual machine in the sandbox resembles the hosts on the network it protects to the greatest extent possible. For instance, assume that malware bound for a host on a network first reaches the sandbox, which begins to analyze the file. However, since the sandbox is poorly configured and does not resemble the hosts on the network enough, the malware manages to distinguish the sandbox from the hosts and hides its malicious behavior. The sandbox reports the file as benign in good faith, forwards the file to the host on the network it was intended to, which gets infected since the malware detects that it is no longer being executed on a sandbox. In a scenario like this, the more the virtual machine image inside the sandbox resembles the systems on the network, the harder it is for malware to distinguish between the two. Images that are "best effort configured" to protect an environment are often referred to as *golden images*.

There are typically two ways to analyse what happens to a system when a program is run, which apply to sandboxes as well. In the first approach, a system snapshot is taken before and after a program is run, analysing what difference there is between them and what changes have been made to the system. The second approach is to study the program during execution with *hooking* and debugging, which generates a more detailed result than the snapshot approach [34]. Hooking means intercepting function calls (to external libraries, such as DLL:s) and reroute them to customized code called hooks. By utilizing hooks, sandboxes can control function calls commonly made by malware, such as those to the Windows API.

As sandbox products gained popularity a couple of years back, malware developers started implementing countermeasures in the form of anti-analysis techniques. For instance, many malware use anti-virtualization and anti-debugging techniques to detect if they are being analysed, and although the sandboxing technology continuously evolves, so does the anti-analysis techniques [24]. Since sandboxes make use of virtual machines and an average host does not, it is crucial for the sandbox to be able to disguise itself as a "normal" host, i.e. hiding all signs of a virtual machine to the malware being analyzed. As stated in section 1.1, 28 percent of all malware in 2014 was virtual machine aware [3].

2.7.1 Detection of virtual machines and sandboxes

Machines that run hypervisors can dedicate only a limited part of their hardware resources to each guest, since they need the majority of their resources for themselves or for other guests; most times, machines that run hypervisors intend to have more than one guest, since there is little point in having a machine hosting only one guest. Instead, multiple guests often reside on a common host, which means that these guests must share the hardware resources of the host. Since hardware resources are limited, each guest only gets its fair share of disk size, memory and processing power [30]. As a consequence, the resources of guests are generally low - even in comparison to mediocre desktop systems.

Limited hardware is not the only common characteristic among virtual machines; in the guest operating system - which is installed by the hypervisor - there may be numerous traces left by the hypervisor. For instance, the names of hardware or hardware interfaces are often set in such a way that it indicates the

presence of a hypervisor. Furthermore, there may be "helper" processes running in the guest to facilitate e.g. user interaction, and if the guest is a Windows system there is often an abundance of traces left in the Windows registry as well [35].

Sandboxes, just like virtual machines, have characteristics on their end as well. Since an analysis in a sandbox has to be finished within a reasonable amount of time, sandboxes have an upper time limit defining how long a file should be analyzed at most. This time limit is typically only a few minutes.

From a malware developer's perspective, the time out means that simply making malware wait or "sleep" for a duration during execution could possibly make the sandbox time out before detecting any malicious behavior. Such waits or sleeps could easily be introduced in malware by calling functions available in the Windows' API, making it trivial for malware developers to bypass sandboxes and because of this, many sandboxes contain hooking functionality.

2.8 DLL injection and API hooking

The technique of intercepting calls to an API is called API hooking. API calls can be used for many different purposes, and practically every program that is intended to run on Windows interacts with the Windows API; this is also true for malware. A hook intercepts such a function call, and either monitors or manipulates the result of it. There are several ways to implement hooking, but a common way is to use *DLL injection*. A DLL injection forces the target process to load a DLL which overrides the original functions that the program uses with hooked versions of the functions [34]. In order to inject a DLL, the sandbox must manipulate the memory of the running program. This can be done by altering the *Import Address Table* (IAT), which is a table of pointers to addresses of imported functions. It could be functions in a loaded DLL, which makes it possible to implement a hook by changing the pointers to addresses elsewhere. They could either point to a defined code stub that logs the API call and then forwards the program to the original API, or simply call another function. Another way of implementing hooks is called *inline hooking*, where the the sandbox modifies the entry point of an API call by rewriting the first bytes in the API, making it possible to go to a "detour function" or "trampoline function" as demonstrated in Figure 2.1 [36].

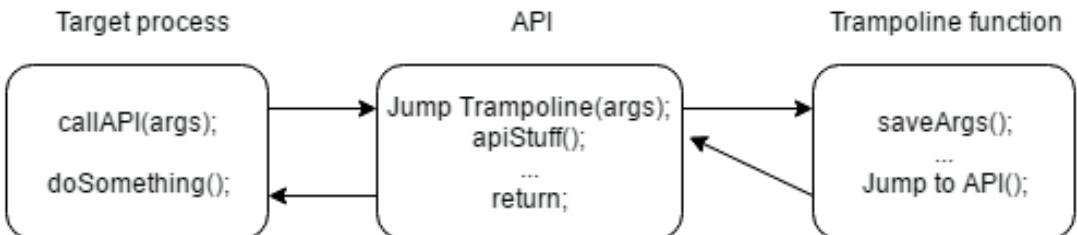


Figure 2.1: Flow chart of an inline API hook.

An essential feature of hooking is that the target process should never be

aware that its function calls are being hooked. Because of this, it is preferable to place hooks as close to the kernel of the operating system as possible, as it becomes more concealed and harder to detect for the target process. The kernel is the core of an operating systems, meaning it has control over everything that happens in the system [37]. A *Central Processing Unit* (CPU) can operate in two modes: kernel mode or user mode. The main difference between the two is that the code being run in kernel mode shares the same address space, while the code run in user mode does not, see Figure 2.2. This means that in kernel mode, a driver is not isolated and could be used and manipulated freely - with the risk of crashing the entire operating system [37] [38].

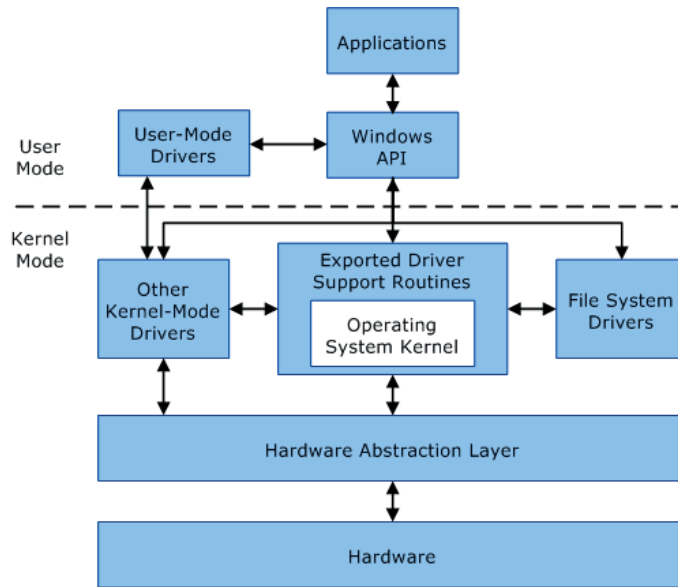


Figure 2.2: Communication between user mode and kernel mode components.

2.9 Sandbox analysis procedure

Security by obscurity, i.e. providing security by keeping a design or an implementation secret, is generally considered bad practice. Despite this fact, most sandbox vendors disclose very little details about their products in order to obstruct the task of performing sandbox detection. Although much of this information could be extracted using test cases similar to the ones in this Master's Thesis, there is still some functionality which remains unknown. Furthermore, since this Master's Thesis neither focuses on confirming the exact functionality of the sandboxes nor aims to do the sandbox vendors a disservice by disclosing details about their products, the exact functionality of the sandboxes is not discussed in great detail. Besides, the behavior and functionality of the different sandboxes differ, and digging into details about each and every one of them is out of the scope of

this Master's Thesis. However, the core functionality of the sandboxes is based on the same basic idea.

Sandboxes may get access to files to analyze in several ways: for instance, files could be manually submitted to the sandbox by a user, or the sandbox could be configured to intercept network traffic and analyze certain files according to defined policies. The procedure that takes place when a sandbox is about to analyze a file may also differ between different sandboxes, but share a common concept [34]. The sandbox runs an operating system in the bottom which, in turn, runs a number of services: a hypervisor to handle virtual machines, a "main" service controlling the hypervisor and potentially monitoring services for networking or screen capturing. Furthermore, there may be a set of additional services such as a web server handling file submissions, a database for storing samples and analysis results etc.

When the sandbox gets access to a file to analyze, the main service delivers the file to the hypervisor which starts a virtual machine either by booting an image or restoring a snapshot. If the virtual machine has DLL:s containing hooked functions, these DLL:s are loaded into memory. Furthermore, if there are monitoring services running inside the virtual machine, these are started. Thereafter the file to be analyzed is executed, and lastly the virtual machine is closed either when execution finishes or when the sandbox times out [34].

The monitoring services, i.e. the services that monitor the behavior of the analyzed file, may run on the host operating system of the sandbox or inside the virtual machine or possibly both. Their purpose is to observe the behavior of the file: what changes it makes to the file system, what processes it creates, what network traffic it generates and so forth. As an example, changes made to the file system could be checked either by the main service in the host operating system by comparing the virtual machine states before and after execution of the file has finished and the virtual machine is shut down, or by a process running inside the virtual machine during execution. The two approaches come with both advantages and disadvantages: although comparing the virtual machines after execution has finished never could be detected by the analyzed file, assuming it uses some sandbox detection technique(s), it lacks the possibility to e.g. detect created files which are deleted before execution finishes [34].

After the monitoring services have finished, their results are put together to form a common score based on a set of criteria. This score is the final result of the analysis, and its value determines the judgement of the sandbox regarding whether or not the analyzed file is malicious. The criteria on which the score is based is not disclosed by different sandboxes and likely differs a bit between them. However the score of each action taken by the analyzed file, is based on how malicious the sandbox deem the action. Typically there are some actions that are a very clear sign of malicious behavior, e.g. changing a Windows Registry, modifying the Windows Firewall or connecting to a known malicious domain. If similar actions are taken, the sandbox can with good confidence verdict the file as malicious. On the other hand there are actions that are not as malicious but a series of less malicious actions could still be combined to do something evil. This combination should the sandbox be able to track and verdict the file as malicious. Many sandboxes typically have the functionality to drop all the files created and

isolate them. They can also monitor newly created processes and keep track of them as well. If a spawned process or file is malicious the original file should of course also be considered malicious.

2.10 Related work

This Master's Thesis is not the first experiment in the area, as there were a few papers that lead Coresec to the idea of this project. In 2014, Swinnen et al. suggested different, innovative ways of packing code, one of which were adopted by the authors of this Master's Thesis and very much inspired them in the design of their launcher malware [1]. They also presented a handful of detection techniques, some of which were utilized in the experiments of this Master's Thesis

Singh. et al. showed that as early as three years ago, sandbox evasion started becoming a threat (although sandboxing was quite a new technology by the time) [39]. The authors present a number of test categories with test cases, which are similar to the ones developed in this Master's Thesis.

In 2015, Balazs developed a tool that mines sandboxes for information [40]. His findings served as a source of inspiration to some of the test cases in this Master's Thesis.

Lastly, there are a few more papers, such as those by Gao et al. and Vasilescu et al., which are similar to this Master's Thesis but less comprehensive [41] [42].

3.1 Introduction

The vendors that participate in this Master's Thesis have in common that they provide a dynamic anti-malware system, i.e. a sandbox. Furthermore, they are all considered to be the top-of-the-line vendors within the sandboxing industry. All vendors were guaranteed absolute anonymity in this Master's Thesis, due to the fact that publishing details on how to bypass their sandboxes could harm both the vendor directly and Coresec indirectly, who is a partner to many of the vendors. Therefore, the vendors are denoted from A - E.

Two different options exist for setting up the sandboxes in the network and giving them access to the files to be analyzed. The first option is to configure the sandbox to listen to network traffic, and automatically detect, fetch and analyze files according to certain criteria such as file type. The other option is to simply upload files to be analyzed manually using a graphical user interface (GUI). In this Master's Thesis, these two setups are referred to as *inline* and *upload* respectively, and details on how each sandbox was configured can be found in section 4.3.

3.2 Vendors and sandboxes

3.2.1 Vendor A

Vendor A has a strong background within the firewall industry, and although it now provides several different systems - a sandbox being one of them - forming a full platform, the firewall(s) are still regarded as the foundation of the security platform. Vendor A is one of the three medium-sized vendors in this Master's Thesis, see Figure 3.1, and has been in the business for roughly a decade, see Figure 3.2.

Vendor A's sandbox

Vendor A has been involved with sandboxing since 2011, which was also the year when Vendor A released the first version of its current sandbox. The sandbox is

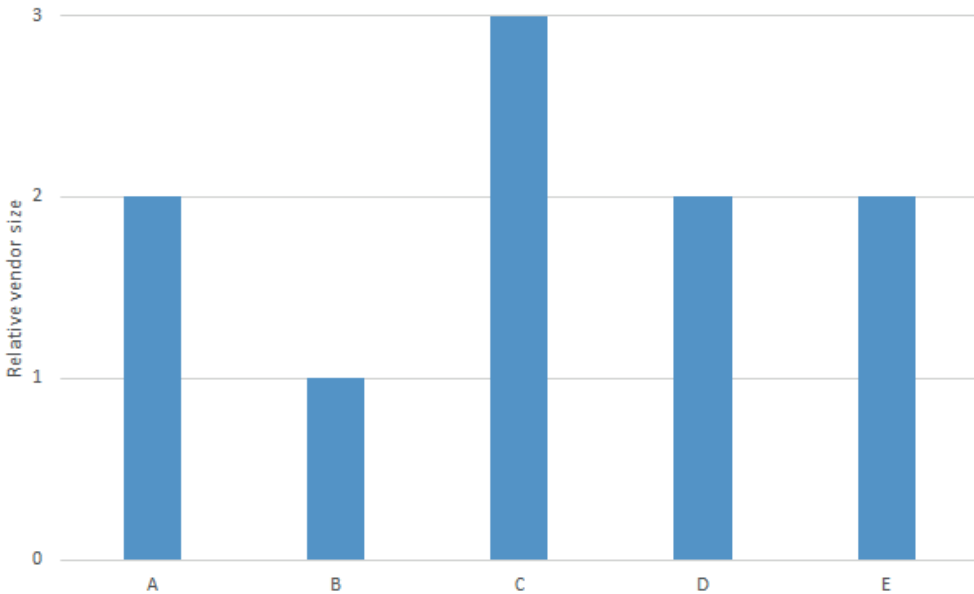


Figure 3.1: Relative vendor size measured in number of employees

available both as a virtual machine and as a physical appliance and it supports images running either Windows XP or Windows 7. However, it does not support the use of golden images, but Adobe Reader, Adobe Flash and the Microsoft Office suite are installed by default. The actual sandbox is located on Vendor A's servers in the cloud, and the physical appliance or virtual machine that is installed in the network of a customer is responsible simply for fetching files to analyze - either in inline mode or by upload - and delivering them to the cloud servers. Therefore, an Internet connection is required by the appliance in order for the analysis to take place. When malicious files are detected, signatures are generated and shared with the threat cloud network of Vendor A within minutes.

3.2.2 Vendor B

Vendor B was founded in the mid 1990s and has therefore been in the business for twice as long as Vendor A. Despite this, Vendor B is the only small-sized vendor of this Master's Thesis with half as many employees as Vendor A. The primary focus of Vendor B is web security.

Vendor B's sandbox

Vendor B has been doing sandboxing since the mid 2000s but the current sandbox of Vendor B was released as late as in 2013. The sandbox is only available as a physical appliance, and it supports images running Windows XP, 7 or 8. These images come in a number of variants which differ regarding installed software. The sandbox should be Internet connected for the purpose of threat intelligence

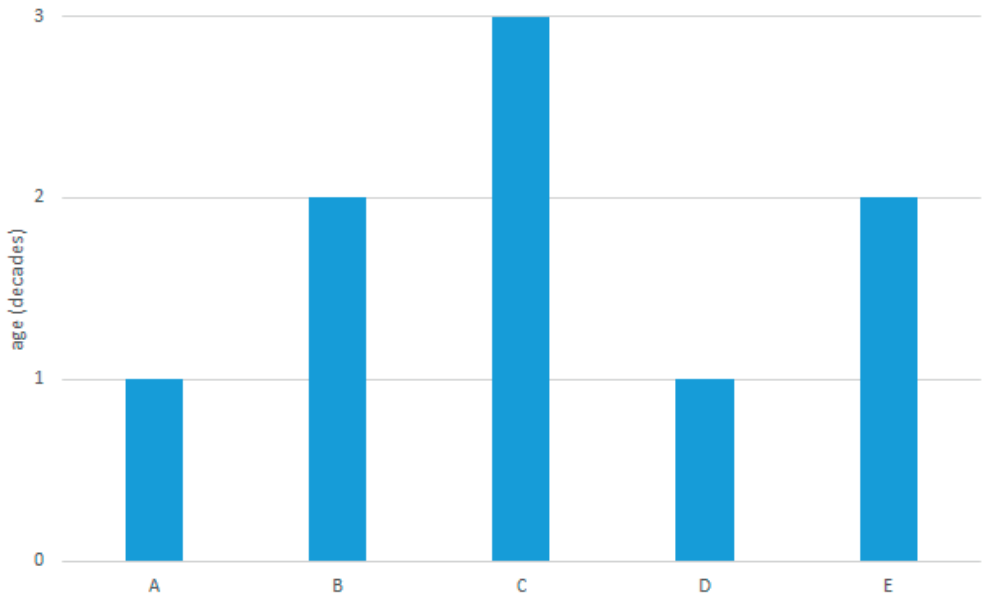


Figure 3.2: Relative vendor age

sharing with Vendor B's cloud, but the analysis environment - i.e. the virtual machine(s) within the sandbox - does not get Internet access during the analysis.

3.2.3 Vendor C

With its roughly 5000 employees and three decades within the business, Vendor C is both the oldest and the biggest of the vendors. Vendor C has historically focused primarily on endpoint protection systems (such as traditional antivirus) targeting both business and private users. Therefore, Vendor C is different from the other vendors in the sense that it has a less corporate dominated customer base. However, Vendor C today has a wide spectrum of products, a sandbox being one of them.

Vendor C's sandbox

Vendor C's sandbox was released in 2012 when Vendor C got into the sandbox business. It can be purchased as a physical appliance or as a virtual image just like Vendor A's. In terms of specification and features, the sandbox of Vendor C resembles Vendor B's but stands out in the crowd a bit more. Just like Vendor B's sandbox, it supports images running Windows XP, 7 or 8. However, in addition to these desktop operating systems, it also supports the server operating systems Windows Server 2003 and 2008. Adobe Reader, Adobe Flash and the Microsoft Office suite are installed by default on all images. In addition to this, the sandbox supports the usage of golden images, being the only one except Vendor B's sandbox to include this feature. The sandbox has the option of choosing whether or

not to allow its virtual machines to have Internet access during the analysis, but the sandbox itself lacks any kind of cloud intelligence sharing which exists for all other sandboxes.

3.2.4 Vendor D

Vendor D is the youngest vendor together with Vendor A, having about the same number of employees as well. Sandboxing is one of the main focus areas of Vendor D, and their sandbox therefore serves as a core component on which many of their other products rely.

Vendor D's sandbox

Although Vendor D is young in comparison to the others, it has been doing sandboxing since the start in the mid 2000s. What makes Vendor D's sandbox different compared to the others is its relatively limited set of features. First of all, Vendor D's sandbox only comes as a physical appliance and is not available virtually, in contrast to all other sandboxes but Vendor B's. Furthermore, it only supports images running Windows XP or 7, and it does not support golden images. On top of this, Vendor D discloses no details about which software is installed on their images, and the virtual machines inside the sandbox do not get Internet access when analyzing samples. However, Vendor D's sandbox supports cloud synchronization to share threat intelligence data frequently.

3.2.5 Vendor E

Just like Vendor A, Vendor E started out within the firewall business during the mid 1990s. Today, Vendor E has evolved from being only a firewall vendor into providing a full set of security systems, including a sandbox which is considered a key component.

Vendor E's sandbox

Vendor E got into the sandbox business as late as in 2013. The features of the sandbox of Vendor E is very similar to those of Vendor A and lie close to what could be seen as the "common denominator" of the tested sandboxes. It comes both as an appliance and as a virtual image, it supports Windows XP and 7 and have Adobe Reader and Microsoft Office installed. It does not give sample files Internet access during analysis, but shares threat data in the cloud like most other products do. What differs the most compared to other sandboxes is probably its age: it was released as late as 2015.

3.2.6 Sandbox specification summary

Table 3.1 shows a specification of all sandboxes used in the experiment.

Table 3.1: Sandbox specifications and features

	Vendor A	Vendor B	Vendor C	Vendor D	Vendor E
Physical or virtual	Both	Physical	Both	Physical	Both
Supported Windows images	XP 7	XP 7 8	XP 7 8 Server 2003 Server 2008	XP 7	XP 7
Golden image support	No	Yes	Yes	No	No
Internet access	-	No	Yes	No	No
Cloud synchronization	Yes	Yes	No	Yes	Yes

4.1 Introduction

The term sandbox detection is a generic name used in this Master's Thesis to describe both sandbox detection and virtual machine detection. Although in reality these are two separate areas with their own characteristics, they are treated as one in this Master's Thesis due to the following facts:

- All sandboxes are running on virtual machines.
- Few users run virtual machines.

Effectively, this means that no matter if a sandbox or a virtual machine is detected, the test case assumes that it is being executed on a sandbox and exits. In other words, no distinction is made in this Master's Thesis between sandbox detection and virtual machine detection; they both go under the name "sandbox detection".

All of the vendors stress the fact that their sandboxes should not be deployed as stand-alone systems but as part of a full security suite or platform in order to reach their full potential. This is due to the fact that sandboxes alone are less powerful than sandboxes in combination with other tools or systems performing other types of analyses. Despite this, since this Master's Thesis focuses on detection and evasion of sandboxes only and not full security suites, no complementary security systems are deployed.

4.2 Test case design

The experiment consists of 77 test cases which are made up of two main components: a *launcher* (see section 2.1.2) written by the authors of this Master's Thesis, and a well-known malware which has been around since 2013. The launcher in turn consists of a few components, the most essential ones being a sandbox detection function, a resource loader and a decryption function, see Figure 4.1. The sandbox detection function utilizes a sandbox detection technique, and the malware component is executed only if the sandbox detection concludes that the file is not being executed in a sandbox, see Figure 4.2. All test cases use the same malware, but the sandbox detection function is unique to each test case. The purpose

of the test cases is to determine which of the sandbox detection techniques that manage to successfully identify sandboxes while avoiding to reveal themselves and being classified as malicious by the sandbox.

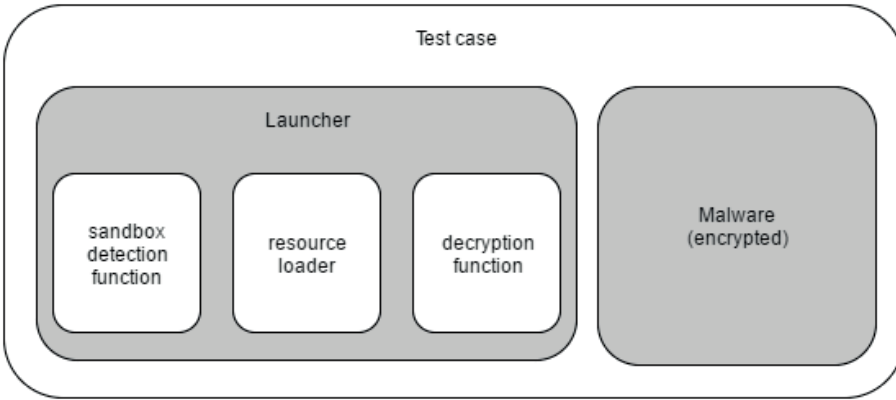


Figure 4.1: Test case components overview

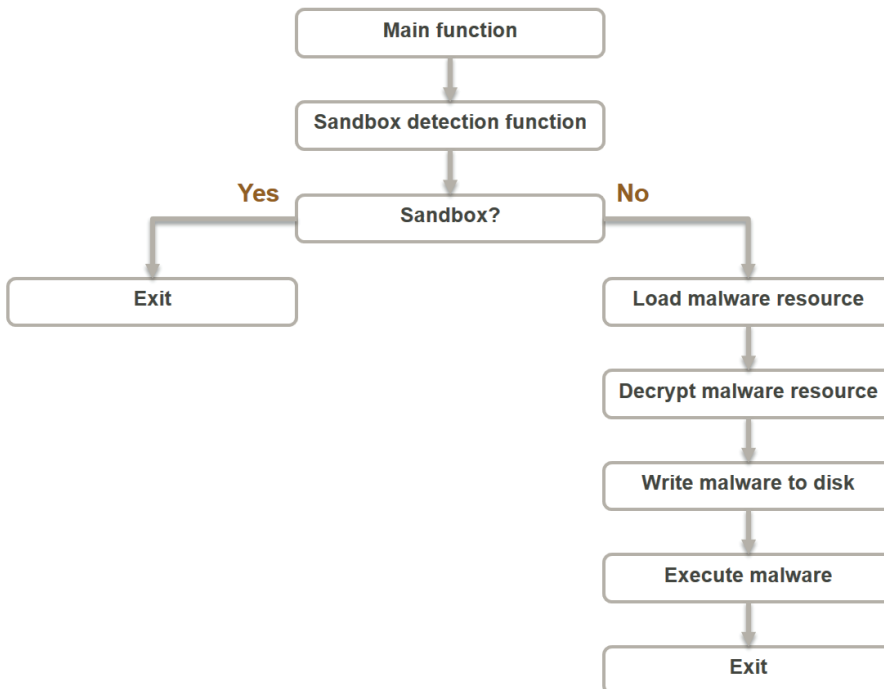


Figure 4.2: Test case flow chart

For the experiment to be as realistic as possible, it was required that the test cases developed would actually execute on the majority of average hosts, otherwise the result would be of little value. To clarify, test cases that detect and

bypass sandboxes are rather meaningless if they do not execute on the systems of "normal" users; the major challenge in developing good test cases lies in distinguishing sandboxes from hosts. An obvious issue related to this is that hosts may be set up in an infinite number of ways, and it is safe to assume that more or less every host that exists will have a unique configuration. Therefore, in order for the test cases to be capable of making this distinction, some assumptions had to be made regarding a typical host configuration. Since most sandboxes are used to protect corporate networks, the assumed environment was a typical, average workstation in a corporate network. The following is a selection of the hardware criteria assumed to be met by hosts:

- CPU: > 2 (logical) cores, 4-16 MB cache
- Memory: >= 4 GB
- Hard disk: >= 100 GB
- Domain connected with Internet access
- (Network) printer(s) installed

In addition to the hardware specifications, a number of assumptions regarding the software were made as well; these details can be found in section 4.4. While some of the assumptions are just educated guesses, others are based on brief studies and random samples from e.g. computers available to the authors via Coresec.

The amount of information that was possible to extract about the specifications and features of the sandboxes differed between the vendors. Therefore, in order to make the tests as fair as possible, the sandboxes were treated as "black boxes" and the test cases were designed and developed without consideration to product specific information. Furthermore, to be able to make a direct comparison of the test results of the different sandboxes, it was required that identical test cases were ran on all sandboxes.

A critical aspect of the performance of the sandboxes that may greatly affect the test results is how they are configured. Since the configuration possibilities differ between the sandboxes, configuration could be a major source of error if not done properly.

4.3 Test method, environment and configuration

Four (virtual) machines were installed in the test environment in addition to the sandboxes:

- the **Malware jumpstation**, which ran Windows 7 and was accessed via Remote Desktop Protocol (RDP) sessions. The jumpstation was, as the name implies, only used to be able to access other machines in the test environment.

- the **Malware client** - also a Windows 7 machine - was accessed via the Malware jumpstation and used to deliver the test cases to the sandboxes, either via inline download or direct upload. The Malware client had a separate networking interface for each sandbox, which facilitated testing of different sandboxes by simply enabling and disabling interfaces.
- the **Malware server** ran the Linux distribution CentOS and a web server, which hosted all test cases and made them available for download to the Malware client.
- the **VPN Gateway** was a Linux machine running the Remnux distribution. Its purpose was to provide a VPN connection out from the test environment to the Internet.

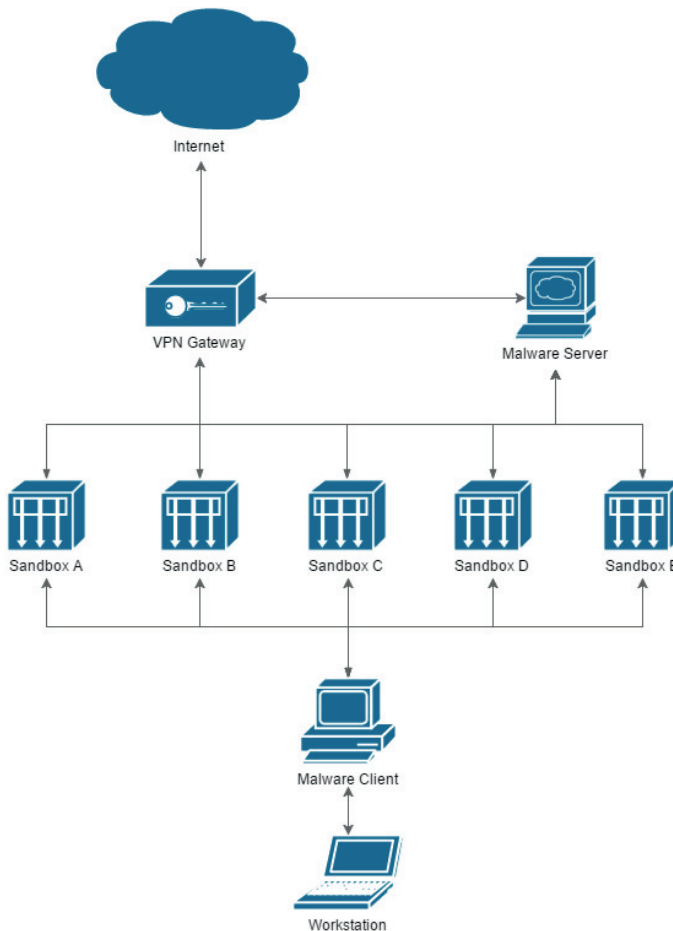


Figure 4.3: A simplified overview of the test environment.

Figure 4.3 shows a simplified overview of the test environment (note that the *Malware jumpstation* is not included in this overview). When running the tests,

the user controls the *Malware client* which downloads the test cases from the Malware server. Since all systems could not be deployed in the exact same way, their actual locations in the test environment differ. However, this does not affect their capability of detecting or analyzing malware; it merely affects how each sandbox gets access to the files to analyze.

Due to the facts that potent malware were to be used in the experiments and that the test environment is located in-house in the headquarters of Coresec, the requirements on the design of the test environment, and especially its networks, were very high; malware ending up in the wrong network and infecting hosts could have very severe consequences. As a precaution, the test environment network was strictly separated into multiple virtual local area networks (VLAN) by a firewall, and access rules between the different VLANs were set explicitly. Each VLAN represented a "zone" with a certain purpose and security level as follows:

- in the **Management zone**, all management interfaces of the sandboxes were placed. The *Malware jumpstation* was also placed here which was accessible via VPN and RDP. This enabled the establishment of a connection to the test environment without physical access to it, which facilitated configuration of the sandboxes.
- in the **Trust zone**, the *Malware client* was located and connected to the sandboxes deployed inline.
- the **Mistrust zone** was set up to allow connections out from the test environment to the Internet and direct this outbound traffic via the *VPN gateway*.
- in the **Demilitarized zone (DMZ)**, the *Malware server* was placed, which hosted all test cases to be downloaded by the *Malware client*.

Figure 4.4 shows the test environment with all zones in greater detail.

To control the Malware client, users first connect to the Management zone via VPN. Thereafter, the user establishes a remote desktop connection to the Malware jumpstation in the management zone, and then further on to the Malware client in the Trust zone.

4.3.1 Inline deployment

The sandboxes of Vendor A, Vendor D and Vendor E were all placed inline, since they had the ability to monitor traffic. Additionally, Vendor A's and Vendor E's sandboxes had routing functionality and were set up with one interface in each zone, meaning that the *Malware client* could connect to the *Malware server* through them and enabling the systems to intercept the transmitted files. The sandbox of Vendor D had no routing functionality, which forced the Malware client to have an interface physically connected to it which in turn was directly connected to the DMZ to reach the *Malware server*.

The sandboxes of Vendor A and Vendor E perform no local analysis; instead, they forward all suspicious files to a global, cloud based analysis center. Due to this, the configuration possibilities of these sandboxes were very limited. In

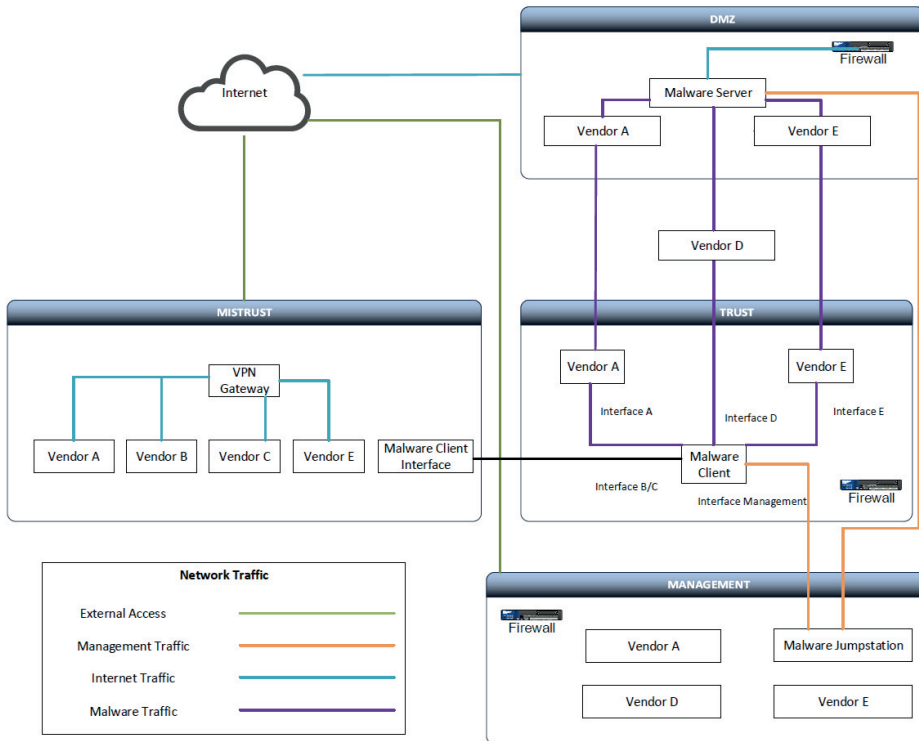


Figure 4.4: Detailed view of the test environment including network interfaces.

the case of Vendor D, the actual analysis took place locally on the sandbox itself, which had three different images all. Despite this, there were not possibilities for the authors to configure or customize these images.

4.3.2 Upload deployment

Vendor B's and Vendor C's sandboxes required files to be manually submitted for analysis, hence they were placed in the Mistrust zone. The Malware client had an interface in that zone as well, and could thus access the GUI of the sandboxes of Vendor B and Vendor C to upload test cases. Both of these sandboxes had more customization possibilities than the others, as they were the only ones who supported golden images. Furthermore, both of them had two images: one with the default settings forced by the vendor and one golden image. The golden images were configured to have higher screen resolution, several non-default programs installed, browsing history, Swedish keyboard layout, additional hard drive partitions and a custom background image.

On the sandboxes of Vendor B and Vendor C, test cases were run both on the original image and golden image to be able to study the potential impact on the results caused by the golden image. However, the result which was considered the official one of these sandboxes was that of the golden image.

4.4 Test cases

All test cases were written in C++ and/or x86 assembly in Microsoft Visual Studio 2015 and compiled into separate, executable files for each test case. A vast majority of the test cases interact with the Windows API or "WinAPI". In some cases there are multiple test cases covering the same area using different techniques, in order to potentially detect if different ways of performing the same task may generate different results due to some functions being hooked by the sandbox and others not.

Since the test cases were developed and compiled on the desktop systems of the authors, the malware could not be included in the test case before or during compilation as this would have required the malware to be placed on these machines. Besides, since the test cases were repeatedly test run on these desktop systems, having included the malware at this stage would have resulted in it being executed and infecting the systems thousands of times during the development, causing loads of wasted time. Additionally, Coresec expressed a wish of developing a service for testing purposes in the future where users could dynamically combine any of the test cases with a set of different malware, which meant that the test cases would have to be separated from the malware. Because of this, a small application was developed which took two files as parameters and added one of them as a resource in the other. How this is done is demonstrated in Figure 4.5, where the "inner" PE file (the malware) is placed in the resource (.rsrc) section of the "outer" PE file (the test case) (see section 2.4 for more details on resources). There is also a "stub" in the .code section of the outer PE file, which is the piece of code responsible for loading the malware resource if a sandbox is not detected.

Once all test cases were compiled, they were moved to an isolated, virtual machine and with the help of the other application they were "loaded" with the XOR encrypted malware as a resource according to Figure 4.5.

All test cases were loaded with the same malware, which was simply an executable file with confirmed malicious behavior. The reason for XOR encrypting the malware was the fact that all sandboxes recognized it as malicious, and since some sandboxes also included tools performing static analysis of the file (which was impossible to deactivate), there was a risk of the test case getting caught in the static analysis. Because of this, the XOR encryption was used to give the malware a new signature that did not already exist in any of the vendors' signature databases.

The test cases consist of a main function, which calls a sandbox detection function representing a test case. If the detection function returns true - indicating the code is being executed in a sandbox - the main function simply exits. However, if the sandbox detection function does not find any signs of a sandbox, it continues. First, it loads the resource containing the XOR encrypted malware. Thereafter, the loaded resource is XOR encrypted again and the original, plain text malware is retrieved. Lastly, the malware is executed and the main function exits. An overview of this process is shown in Figure 4.2.

Some of the test cases are based on techniques that are reoccurring in many articles and have been seen many times before, while others are innovative and completely based on own ideas and inspiration gained during the working pro-

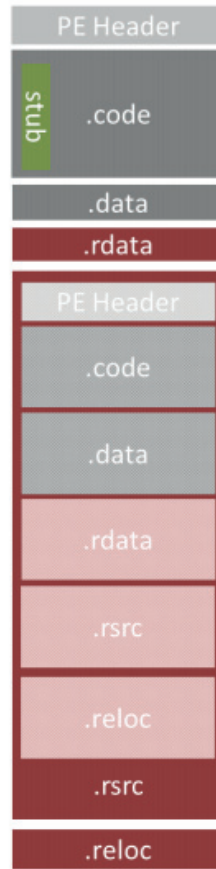


Figure 4.5: A detailed view of how the malware is hidden as a resource in the test case [1].

cess. However, since there is no way for the authors to guarantee that they are actually utilizing a "new" technique for the purpose of sandbox detection, no distinction is made between which test cases are "new" ones and which are not. Instead, the test cases are divided into categories based on what part of the system they focus on as follows:

4.4.1 Timing

The timing based test cases aim mainly to exploit the fact that sandboxes only run the file a limited time, and may have to speed up the program execution in order to finish the analysis before the time-out. In order to achieve this, the test cases take one of two approaches: they either try to stall or slow the execution down, or try to detect that time is being tampered with.

Slowing down the execution is done in two different ways: by sleeping (or waiting) and by using stalling code. Sleeping is achieved simply by calling the

Windows API function `Sleep()`. There are test cases trying both one long, continuous sleep, as well as multiple, shorter sleeps to determine if they generate different results. There is also a test case utilizing threads in combination with `Sleep()` in an attempt to deceive the operating system. Waiting is accomplished by for instance using the Windows API function `SetTimer()`. Both sleeping and waiting are passive approaches that consume minimal system resources. The other approach of slowing down execution, stalling code, is all but passive: it performs resource-intensive, time consuming calculations until the sandbox hopefully times out. There are multiple ways of creating stalling code; the test cases in this Master's Thesis do so by for instance calculating prime numbers or calling API functions that are expected to be hooked (which takes extra time compared to non-hooked functions for the operating system to handle) such as `Sleep()` and `GetTickCount()`.

Detecting time manipulation is simply a matter of measuring time between two events separated by a `Sleep()` call and comparing the result to the expected value: if they differ, the system is most likely a sandbox attempting to speed execution up.

4.4.2 Process

The test cases in this category focus on processes running on the target operating system and DLL:s loaded by these processes, as they may both disclose a sandbox. Since there may be monitoring processes running in the sandbox with names disclosing their behavior, there are test cases to check the names of all processes and compare each one to lists of predefined, suspicious names. Similarly there are test cases to check the names of the DLL:s as well, since potentially hooked functions will reside in DLL:s loaded by the processes.

Windows processes have a "parent process", which will normally be "explorer.exe" if the process is started in the graphical user interface. However, the parent process may differ in sandboxes and could be for instance a monitoring process; whether this is the case or not is investigated by a test case. Some sandboxes have also been reported to change the name of the file being analyzed, e.g. to "sample.exe", which is checked by another test case.

The operating systems ran in sandboxes are often clean installations of Windows, which means that the number of installed programs and running processes will be less compared to regular clients; therefore, there is also a test case in the process category checking the number of running processes on the system.

Finally, there are a number of test cases examining studying what happens when DLL:s are loaded. For instance, a sandbox may try to prevent malware samples from loading DLL:s while still giving the sample the impression that the load succeeded. Therefore, there are test cases both to load valid DLL:s to see if they fail unexpectedly and to load invalid DLL:s to see if they succeed although they should not.

4.4.3 File

These test cases use the file system to determine whether the system is a sandbox or a host. This is done using two different approaches: by looking for the

existence of certain files or by attempting to create files and check for unexpected behavior.

Since the sandboxes were treated as black boxes during the tests, no research could be done before or during the test development phase regarding which specific files potentially existed in the different sandboxes. Instead, the tests had to take the opposite approach and look for files that, with high probability, only would exist on a system that had been used by an actual user, meaning that they would not be present on a sandbox. This could be files related to e.g. commonly installed programs or Internet browsing. The number of installed programs are examined by a test case utilizing the Windows registry, as well as the number of recently modified files - which are expected to be none at a sandbox. As some sandboxes had been reported to run the malware sample from quite untypical directories, the execution path of the sample was also covered by a test case.

Lastly, as there were reasons to suspect that the sandboxes somehow tamper with the file system, a simple test cases exists that creates files in the execution directory, writes content to it and then closes it, and then verifies this content by opening and reading the file afterwards.

4.4.4 Environment

The environment category is more general than the others in the sense that it contains test cases related to the system as a whole rather than specific parts of it. The majority of the test cases are based on spontaneous ideas that emerged either during the background research or during development.

For instance, there are two test cases which examine the names of the machine and the currently logged in user. If any of the names contain certain strings such as "sandbox" or "sample", the likelihood of being on a sandbox is considered to be high. There is another test case checking if there is a password set for the currently logged in user account, as corporate users are expected to have password protected their accounts while sandboxes most likely have not. Furthermore, real users are expected to change the background image and increase the screen resolution from the default value; these parameters are also investigated by one test case each.

The other environment test cases check e.g.:

- The length of the execution path, as execution paths that contain only one or two slashes (such as "C:/" or "C:/temp/") are not very likely to be used by "normal" users,
- Suspicious names in the execution path like "C:/Windows/malwaresample/",
- The number of hard disk drive partitions,
- If any well-known hypervisor is present using different entries of the Windows registry,
- If a debugger is present using the `IsDebuggerPresent()` API function,
- If the system has a battery, since sandboxes will seldom or never be ran on laptops, and

- If the reported system time is somewhat correct (checking against a hard coded date).

4.4.5 Hardware

The hardware test cases inspect various hardware components available to the sandbox. Since the hardware is given to the sandbox by a hypervisor, the hardware test cases are essentially about virtual machine detection rather than sandbox detection. However, since all sandboxes are running on virtual machines and very few users use it in comparison, the number of false positives was expected to be low.

The CPU is very much a limited resource which cannot be extended as easily as hard disk size or amount of memory. The type of CPU, its amount of internal memory (cache) and number of cores are all properties examined by test cases. The amount of memory available to the system and to the graphics adapter (GPU memory), as well as the size of the hard disk drive(s), are other artifacts being tested. All of these are resources expected to be higher on a user client than on a sandbox running inside a virtual machine.

Traces of a hypervisor may not only be found in the hardware alone but in the software closest to the hardware as well. As an example, the name of the graphics adapter interface is assigned by the hypervisor and therefore often contains clues about the presence of a hypervisor. The same is true also for hardware drivers; they may disclose either a hypervisor or the sandbox, if named inappropriately.

Another group of hardware that is not located physically inside the machine is printers. In corporate environments, it is safe to assume that a vast majority will have at least one printer installed and selected as default, while the average sandbox can be expected not to have printers configured. Two test cases are based on this fact.

Finally, there is also a test case to test the presence of a three-button (at least) mouse.

4.4.6 Network

The network category contains test cases checking either properties of network related hardware or actual connectivity status and communication possibilities.

The number of network adapters, the names of the adapters and their (physical) addresses are properties all covered by network test cases. These are based on the following facts or assumptions:

- Clean Windows installations will most likely only have one network adapter, while a user - using for instance a *VPN* (Virtual Private Network) connection or a wireless network - will have multiple adapters. In other words, one network adapter would indicate a sandbox.
- As with the graphics adapters, the hypervisor assigns names to the network adapters and may disclose itself.

- Hypervisors assign certain ranges of *MAC* addresses to network adapters by default, which are well-known. Comparing the *MAC* addresses of the adapters to these ranges is trivial.

The remaining network related test cases check the actual network connectivity. The theory behind this is that some sandboxes may want to prevent malware samples from communicating with e.g. C&C-servers, while still giving the sample the impression of having an Internet connection. Other sandboxes may choose to completely isolate the malware by disable networking. Furthermore, assumptions can be made regarding how the network is configured on an average sandbox compared to how it is done on a typical, corporate user client. Therefore, there are a number of test cases examining what types of connections can be made and which responses are received, to determine if a sandbox is tampering with the network connection. The following is a selection of what is covered by test cases:

- Check if the system is connected to a domain. Most corporate systems will be so, while sandboxes probably will not.
- Check if *ICMP echo requests* (PING) to known hosts receive replies to verify layer 3 connectivity.
- Check if *HTTP GET* requests can be made, and if they receive the expected response.
- Check if *DNS* queries can be made and verify that the responses are as expected.
- Check if a proxy is enabled, which it is assumed to be on considerably more sandboxes than hosts.
- Check how much recent network traffic has been sent and received, which is expected to be far less on sandboxes than clients.
- Check if a socket can be established on port 445, a port used for Windows Active Directory and file shares.
- Check the number of established connections to other hosts, which is also expected to be far higher on user clients than sandboxes.
- Check the presence of other hosts on the same subnet (expected to be none in a sandbox).

4.4.7 Interactive

These test cases have in common that they try to distinguish human interaction from simulated or no interaction. This is done mainly by monitoring input devices - mouse and keyboard - and waiting for certain actions or detecting simulated interaction. The complexity of these test cases varies: the simple ones wait for e.g. a mouse click, mouse movement or a mouse scroll to occur, while the more advanced ones check things such as the speed of mouse cursor (since sandboxes may "teleport" the cursor to click a button or similar), keyboard keystrokes and certain keystroke patterns.

4.5 Test criteria

In order for a test case to be considered successful, two criteria need to be met:

1. the test case must detect the sandbox (and hence not execute the malware), and
2. the sandbox must not classify the test case as malicious.

Although these criteria might give the impression of implying one another, i.e. "if one then two" and vice versa, there is the possibility that the sandbox detection itself function may be classified as malicious. In other words, although a certain test case might be able to detect the sandbox and prevent the malware from being executed, it could still be deemed malicious due to the detection technique used. First of all, the level of discretion of the detection techniques differ: it is reasonable to assume that checking Windows registry keys or executing commands in the Windows shell is considered more "noisy" and raise more flags than checking the screen resolution or disk size. Second, some detection techniques which commonly occur both in articles and in actual malware are likely to have been seen by sandbox developers before (and may therefore be flagged as malicious) while others can be assumed to have been seen less frequently or even not at all. Therefore, a distinction will be made in the result between test cases that fail due to their inability of detecting sandboxes and test cases that fail despite being able to detect the sandbox.

4.6 Choice of malware

The only unconditional requirement on the malware that was used as resource in the test cases was that it was classified as malicious by all sandboxes; otherwise, unsuccessful test cases that failed to detect that they were being run on a sandbox and executed the malware would risk not being detected despite this. Therefore, in order to achieve reliable test results, it had to be assured - before performing the tests - that the chosen malware was indeed identified by all sandboxes. This was done simply by making the malware available to the sandboxes for analysis, either by direct upload or by downloading it through the sandbox depending on the setup.

To maximize the probability of the malware being recognized by all sandboxes, a well-known malware was chosen based on the recommendation of a malware specialist on Coresec Systems. The choice fell on "Beta bot", due to it being both well-known and well-documented. Beta bot is a *Trojan* - a type of malware which disguises itself as other software [43]. The name Trojan emphasizes the appearance rather than the behavior of the malware, and Trojans may therefore be classified as more or less any of the types defined in section 2.1.2.

When executed, Beta bot attempts to gain administrator-level privileges on the target system by presenting a fake Windows "User Account Control" prompt [44]. If the user is fooled and approves the prompt, Beta Bot is able to perform a number of evil actions: it prevents access to numerous security websites, disables local anti-virus software and steals user data. Beta Bot was reported by Symantec

early in 2013, and the severity of Beta bot turned out such that both the Department of Homeland Security (DHS) and the Federal Bureau of Investigation (FBI) to issue warnings regarding Beta Bot [45] [46]. The magnitude of Beta Bot in combination with the fact that it had been around for three years by the time of writing this Master's Thesis made it highly likely that all sandbox vendors would be able to identify it.

5.1 Introduction

Due to the sheer amount of data, the test results are first grouped by category under section 5.2 for better readability. Thereafter, some test data from the golden image sandboxes is presented in section 5.3. A summary of the complete test results is given in section 5.4 at the end of this chapter.

5.2 Results per category

Initially, for each category, statistics are given regarding the average, maximum and minimum malware detection rate (i.e. the rate of unsuccessful test cases). The coefficient of variation (or *relative standard deviation*) is given, which is a measure of dispersion of the results between the different vendors within each category.

The statistics table of each category is followed by a detailed result table. The cells of the result table contain one of the three values *pass*, *fail* and *no exec*. *Pass* means that the test case succeeded in detecting the sandbox while staying undetected, i.e. it succeeded from an "attacker's" perspective. Passed test cases are highlighted with a green background color in the tables. Cells that contain either *fail* or *no exec* are unsuccessful test cases, which either failed to detect the sandbox (*fail*) or were considered malicious by the sandbox despite not executing the malware (*no exec*). The reason for distinguishing these two from one another is that the *no exec* test cases are closer to succeeding than the *fail* ones, since they managed to detect the presence of the sandbox.

Finally for each category comes a bar chart, displaying the number of detected (i.e. the sum of *fail* and *no exec*) test cases per vendor.

5.2.1 Timing

Table 5.1: Timing results statistics.

Average malware detection rate	46%
Maximum malware detection rate	70%
Minimum malware detection rate	30%
Coefficient of variation	33%

Table 5.2: Timing category results per vendor.

#	Description	A	B	C	D	E
1	one long sleep	fail	fail	fail	fail	no exec.
2	multiple short sleeps	fail	pass	pass	fail	no exec.
3	delay using timer	pass	pass	pass	fail	no exec.
4	use two threads to detect sleep emulation	fail	pass	fail	pass	fail
5	create files, verify time stamps	fail	pass	pass	pass	fail
6	open files, verify time interval	fail	fail	fail	fail	fail
7	check emulated sleep with tickcounts	pass	fail	pass	fail	no exec.
8	stalling loop, prime calculations	pass	pass	pass	pass	pass
9	stalling loop, WinAPI calls	pass	pass	pass	pass	pass
10	combine multiple timing techniques	pass	pass	pass	pass	pass

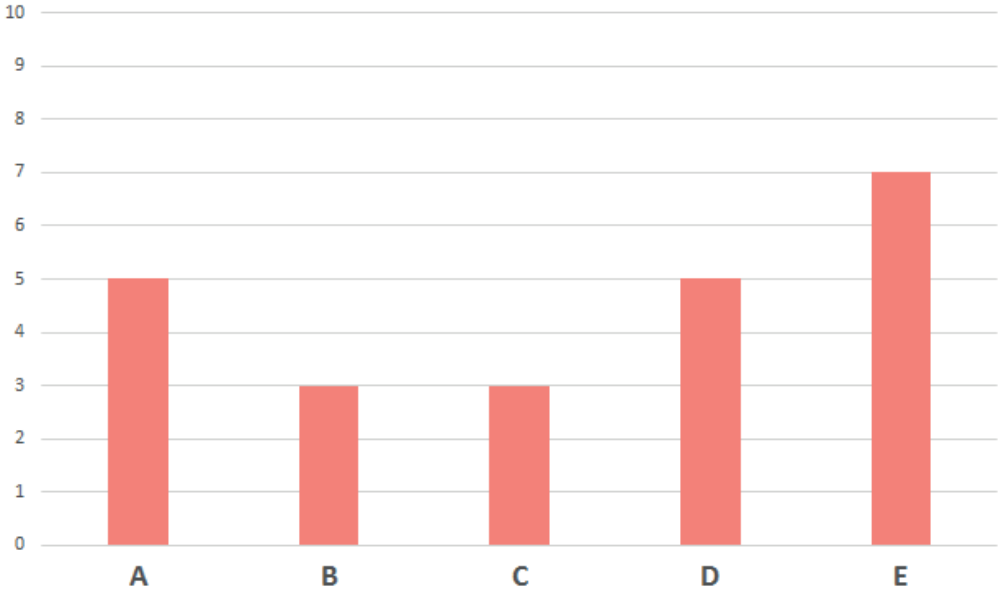


Figure 5.1: Timing - detected malware per vendor.

5.2.2 Process

Table 5.3: Process results statistics.

Average malware detection rate	74%
Maximum malware detection rate	80%
Minimum malware detection rate	60%
Coefficient of variation	11%

Table 5.4: Process category results per vendor.

#	Description	A	B	C	D	E
1	check sandbox processes	fail	fail	fail	fail	fail
2	check client processes	pass	pass	fail	fail	no exec
3	number of processes	pass	pass	fail	pass	no exec
4	name of parent processes	fail	fail	pass	fail	no exec
5	file name for current file	pass	fail	fail	fail	pass
6	load fake DLL	fail	fail	fail	fail	fail
7	load real DLL	fail	fail	fail	fail	fail
8	DLL load directory	pass	pass	pass	pass	pass
9	names of loaded DLLs	fail	fail	fail	fail	fail
10	known sandbox DLL:s	fail	fail	fail	fail	fail

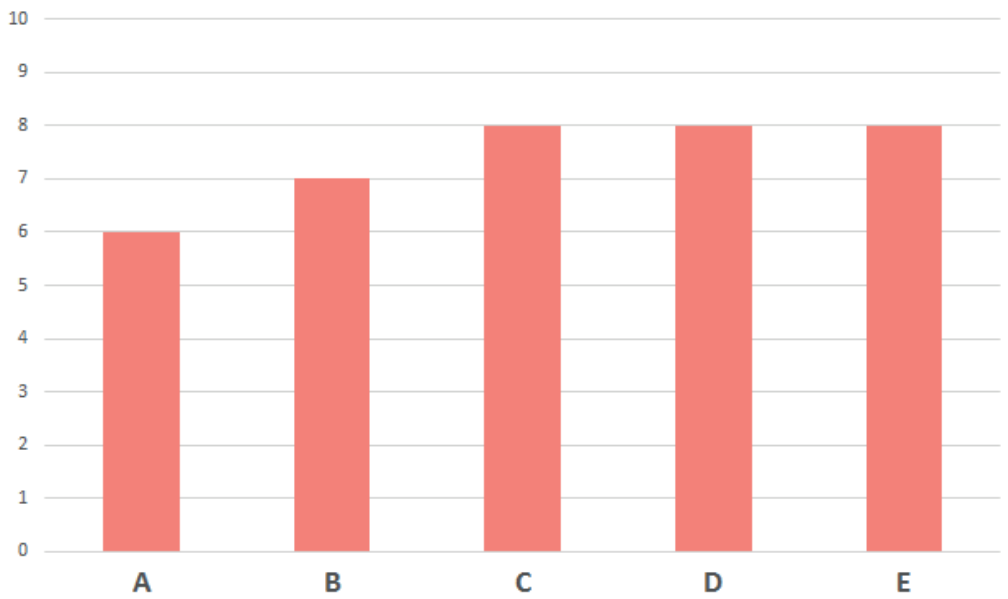


Figure 5.2: Process - detected malware per vendor.

5.2.3 File

Table 5.5: Files results statistics.

Average malware detection rate	73%
Maximum malware detection rate	100%
Minimum malware detection rate	50%
Coefficient of variation	23%

Table 5.6: File category results per vendor.

#	Description	A	B	C	D	E
1	if browsers exist	fail	fail	fail	fail	fail
2	traces left by browsing	pass	fail	fail	pass	fail
3	number of other files in execution directory	fail	fail	pass	fail	fail
4	create files locally and check their content	fail	fail	fail	fail	fail
5	number of installed programs	fail	pass	pass	fail	fail
6	number of recently created/modified files	pass	fail	pass	pass	fail

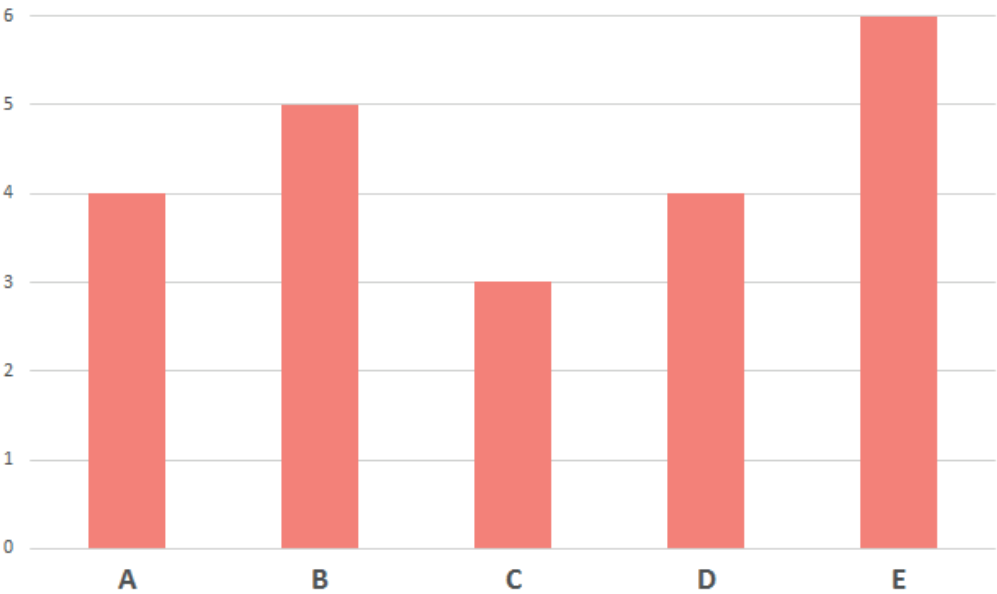


Figure 5.3: Files - detected malware per vendor.

5.2.4 Environment

Table 5.7: Environment results statistics.

Average malware detection rate	64%
Maximum malware detection rate	81%
Minimum malware detection rate	56%
Coefficient of variation	16%

Table 5.8: Environment category result per vendor.

#	Description	A	B	C	D	E
1	machine names	fail	pass	fail	fail	fail
2	user name(s)	pass	pass	pass	pass	pass
3	screen resolution	pass	pass	fail	pass	pass
4	hard disk / partition name(s)	pass	pass	fail	pass	fail
5	color of background pixel	fail	fail	fail	fail	fail
6	keyboard layout	pass	fail	fail	pass	pass
7	execution path	pass	pass	fail	fail	pass
8	sandbox indications in path name	fail	fail	pass	pass	fail
9	command line parameters	fail	fail	fail	fail	fail
10	if computer is a laptop	pass	pass	pass	pass	no exec
11	VMWare version	fail	fail	fail	fail	fail
12	correct system time	fail	fail	fail	fail	pass
13	if current user has password set	pass	pass	fail	pass	no exec
14	is debugger present?	fail	fail	fail	fail	fail
15	typical sandbox registry keys	fail	fail	fail	fail	fail
16	specific key values, if they contain sandbox info	fail	fail	fail	fail	no exec

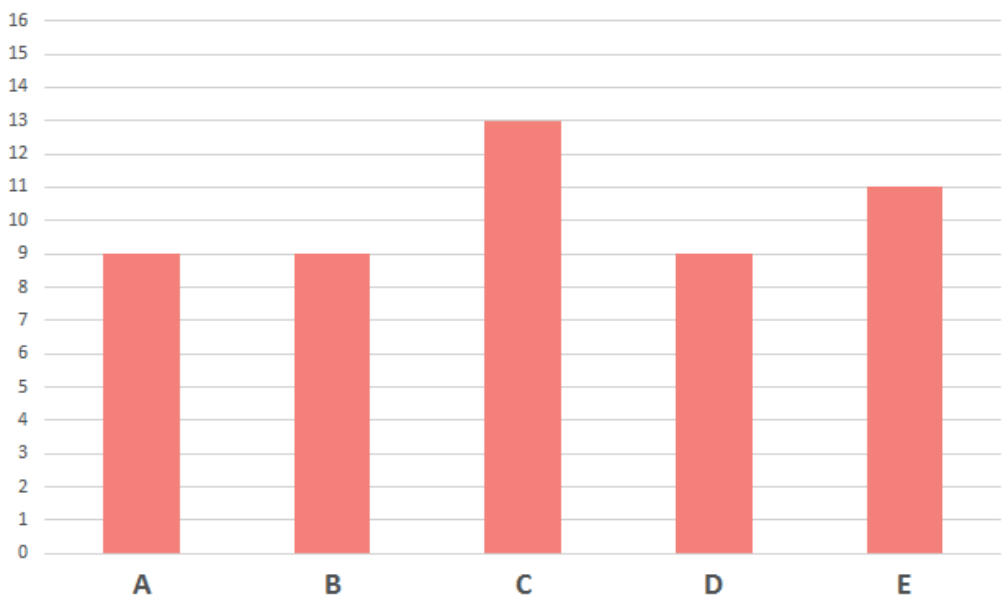


Figure 5.4: Environment - detected malware per vendor.

5.2.5 Hardware

Table 5.9: Hardware result statistics.

Average malware detection rate	49%
Maximum malware detection rate	54%
Minimum malware detection rate	46%
Coefficient of variation	8%

Table 5.10: Hardware category result per vendor.

#	Description	A	B	C	D	E
1	number of CPU cores, assembly	pass	pass	pass	pass	no exec
2	number of CPU cores, WinAPI	fail	fail	pass	pass	no exec
3	number of CPU cores, C++11	pass	fail	pass	pass	no exec
4	CPU type	fail	pass	pass	fail	fail
5	CPU cache size	fail	pass	pass	fail	pass
6	amount of RAM	pass	fail	fail	pass	pass
7	size of disk(s)	fail	fail	fail	fail	pass
8	amount of GPU Memory	fail	fail	fail	fail	fail
9	name(s) of GPU adapter(s)	pass	pass	pass	pass	pass
10	names of loaded drivers	fail	fail	fail	fail	fail
11	number of (local) printers	pass	pass	fail	pass	pass
12	name of default printer	fail	pass	fail	fail	fail
13	presence of 3-button mouse	pass	pass	pass	pass	pass

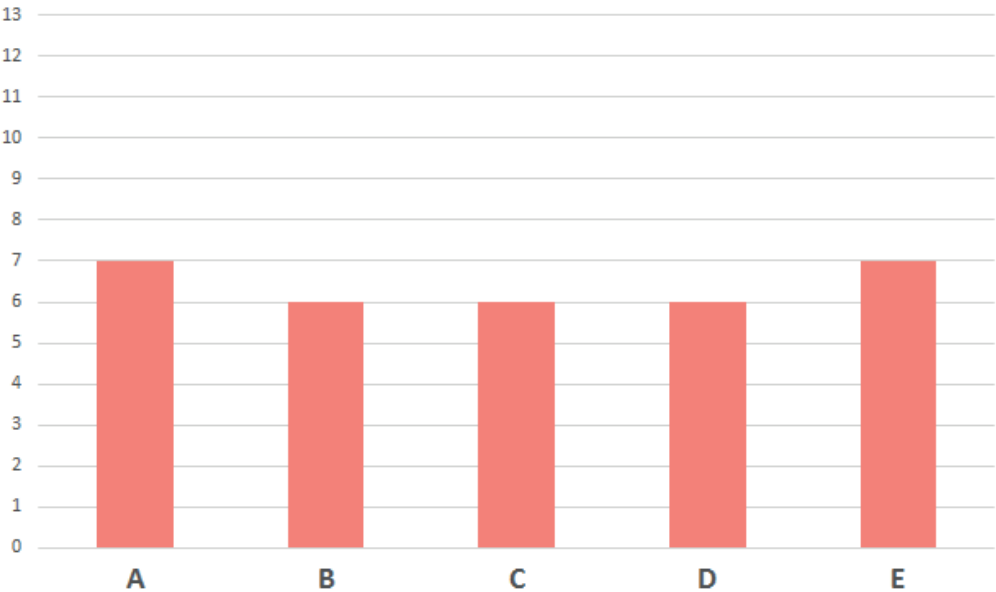


Figure 5.5: Hardware - detected malware per vendor.

5.2.6 Network

Table 5.11: Network result statistics.

Average malware detection rate	46%
Maximum malware detection rate	62%
Minimum malware detection rate	23%
Coefficient of variation	35%

Table 5.12: Network category results per vendor.

#	Description	A	B	C	D	E
1	number of network interfaces	pass	pass	pass	pass	pass
2	name(s) of network interface(s)	fail	fail	fail	fail	fail
3	mac address(es)	fail	pass	fail	fail	fail
4	ICMP (ping)	fail	pass	fail	pass	pass
5	HTTP (GET)	pass	pass	fail	pass	fail
6	correct response from HTTP request	pass	pass	fail	pass	pass
7	DNS query	fail	pass	fail	pass	pass
8	proxy enabled	fail	fail	fail	fail	fail
9	domain connected	pass	pass	pass	pass	no exec
10	open ports	fail	fail	fail	fail	fail
11	network traffic	pass	pass	pass	pass	pass
12	established connections	no exec	pass	pass	pass	pass
13	other hosts in network	fail	pass	pass	pass	fail

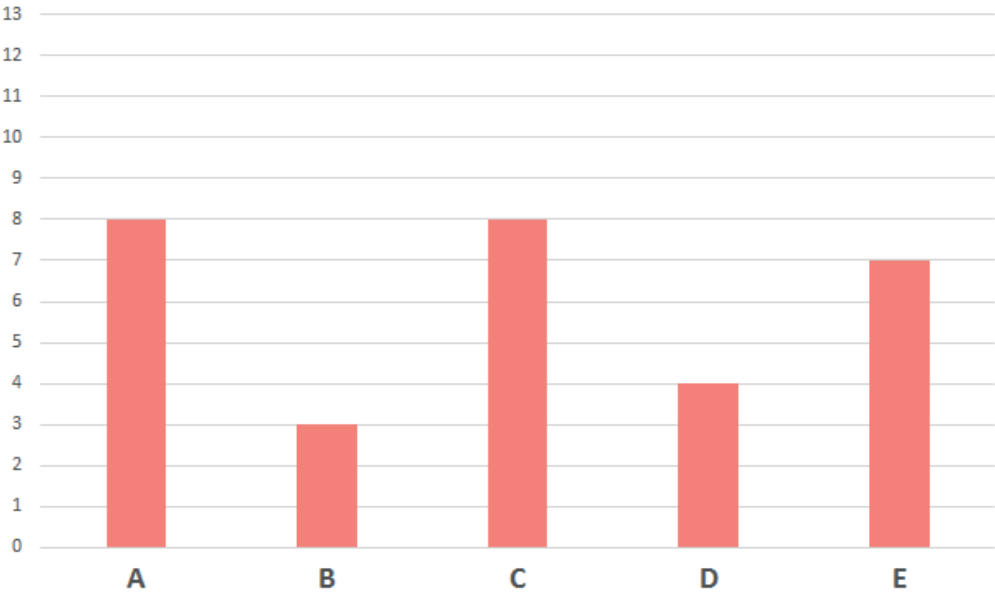


Figure 5.6: Network - detected malware per vendor.

5.2.7 Interactive

Table 5.13: Interactive result statistics.

Average malware detection rate	51%
Maximum malware detection rate	67%
Minimum malware detection rate	33%
Coefficient of variation	22%

Table 5.14: Interactive category results per vendor.

#	Description	A	B	C	D	E
1	mouse clicks	fail	fail	fail	fail	fail
2	mouse cursor position	pass	fail	fail	fail	fail
3	mouse movements	fail	fail	fail	fail	no exec
4	high mouse movement speeds	pass	pass	pass	fail	no exec
5	wait for scroll	pass	pass	no exec	pass	pass
6	multi-choice prompt	pass	pass	pass	pass	pass
7	keyboard interaction (key presses)	pass	pass	pass	pass	no exec
8	keyboard typing patterns	pass	pass	pass	pass	pass
9	no open windows	fail	fail	fail	fail	fail

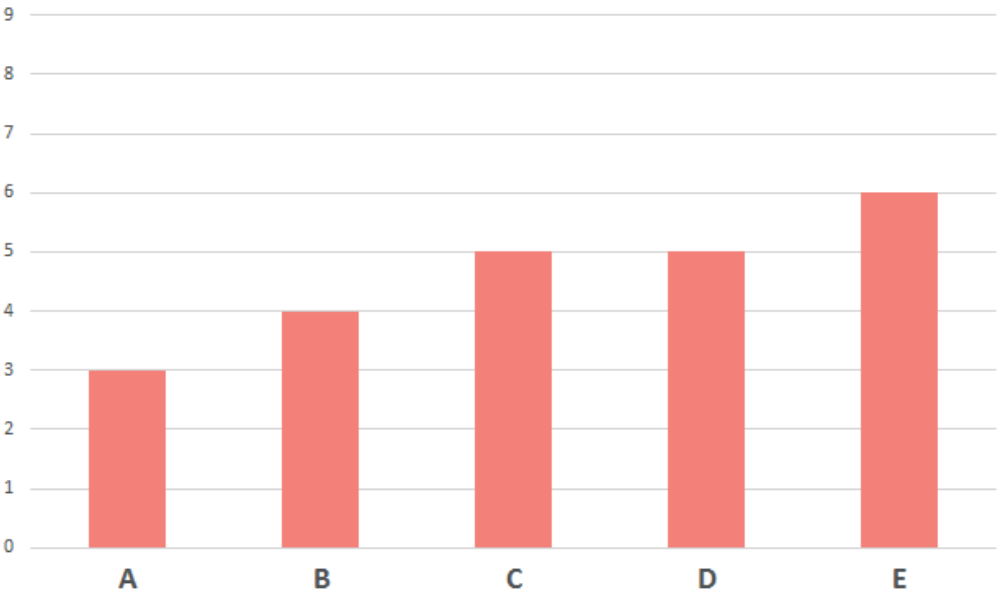


Figure 5.7: Interactive category - detected malware per vendor.

5.3 Result of golden image configuration

In a total of eight test cases, there are differences between golden images and original images. Per vendor there are five test cases but only two common test cases where the golden image has the same outcome, see table 5.15.

Table 5.15: Result between golden and original images.

	B Original	B Golden	C Original	C Golden
size of disk(s)	fail	fail	pass	fail
number of (local) printers	pass	pass	pass	fail
if browsers exist	pass	fail	fail	fail
traces left by browsing	pass	fail	pass	fail
number of recently created/modified files	pass	fail	pass	pass
screen resolution	pass	pass	pass	fail
keyboard layout	pass	fail	pass	fail
no open windows	pass	fail	fail	fail

5.4 Result summary

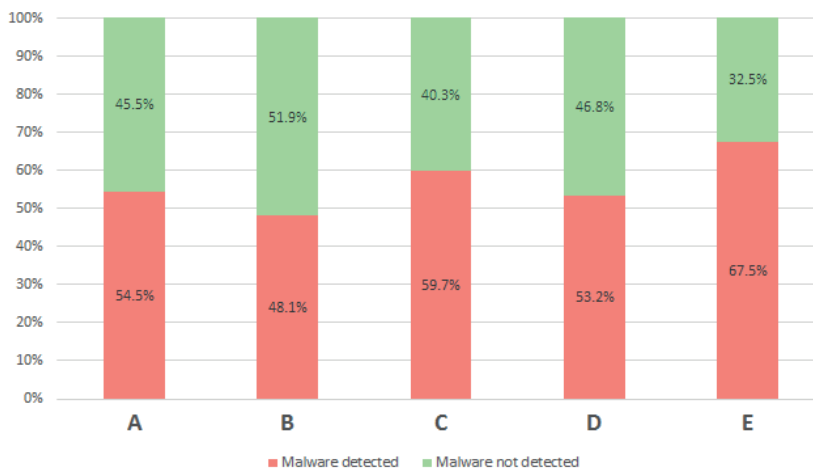


Figure 5.8: Malware detection rate per vendor.

Table 5.16: Overall result statistics

Average malware detection rate	56.6%
Number of silver bullets	11

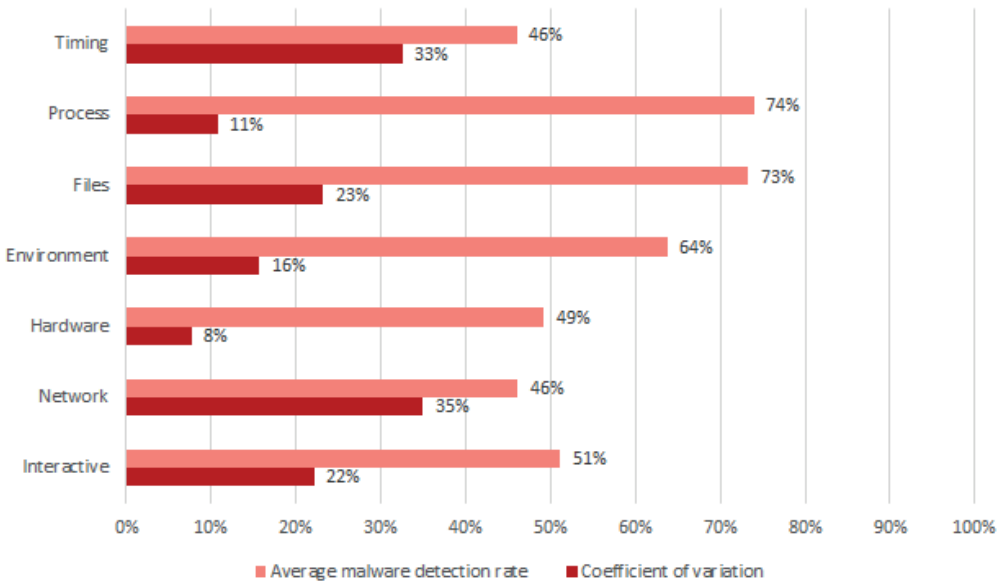


Figure 5.9: Average malware detection rate per category and coefficient of variation.

6.1 Sandbox technologies and result patterns

Drawing conclusions from the results based on the technologies used in the different sandboxes proved to be harder than expected, as the amount of information that was available about the sandboxes was very limited compared to the expectations at the start of the project. The publicly available information in datasheets and similar documentation was consistently of very general nature, and the somewhat detailed information that reached the authors was via people at Coresec who either "heard from someone" or had their own theories or educated guesses - both of which unfortunately are of little value in a scientific article. Consequently, the opposite approach had to be taken to answer this question: instead of drawing conclusions about the result based on the technologies used in the sandboxes, reasonable assumptions were made about the technologies based on the test results.

Looking at the overall result, the top achiever is Vendor E. Vendor E has the best result in five of the seven test case categories, and places second in the remaining two. It detects more than two thirds of all test cases, and is 13% (or 8 percentage points) better than the next best vendor, Vendor C. Except from the impressive test results, another thing distinguished Vendor E from all other vendors: in 17 of the 77 test cases, the sandbox of Vendor E classified the test case as malicious although the test case detected the sandbox and neither decrypted nor ran the malware resource. This only happened very occasionally for the other sandboxes - once for Vendor A and Vendor C- which suggested that something was different with the sandbox of Vendor E compared to the others.

The first theory that emerged as a result of Vendor E's somewhat odd test results was that its sandbox simply had stricter rules regarding malicious behavior and that the amount of false positives consequently would be higher on their sandbox compared to the others. To confirm this theory, the authors tried to generate false positives by uploading various modified samples (e.g. with an encrypted, benign executable file as resource instead of malware) to the sandbox of Vendor E, but surprisingly, every single attempt of doing so failed although quite some effort was put into this. The false positives theory was dismissed, and instead some additional research was made regarding the sandbox of Vendor E and its results. The authors soon came up with another theory, namely that the sand-

box of Vendor E was supported by another tool, containing both static analysis and something called *code emulation*, as some parts of the detailed result reports indicated this. More specifically, there were certain text strings in the reports from the sandbox which resembled static signatures originating from a certain vendor (not included in the Master's Thesis) well known for its anti-virus products. When digging deeper into this, it appeared that there was indeed a collaboration between this vendor and Vendor E. Unfortunately, presenting these signatures in this Master's Thesis would disclose enough information for the reader to be able to figure out the identities of both of these vendors, but they suggested that a static analysis had indeed taken place. However, it seemed highly unlikely to the authors that a static analysis tool alone would be capable of distinguishing the false positives samples from actual positives with such high success rate (with respect to the strong encryption of the malware resource), which suggested that there could be other tools involved as well. When doing more research about the anti-virus vendor and its products, it appeared that their anti-virus contained something called *code emulation* as well, a feature that actually analyzes samples dynamically by executing them but in an emulated environment instead of a full scale virtual operating system like a sandbox. This could explain why this additional tool was capable of detecting test cases while being able to distinguish false positives from actual positives.

Except for this, Vendor E also claims to have a feature in their sandbox which monitors activities on a low system level. More specifically, the sandbox monitors CPU instructions and patterns of instructions, which - if they match certain rules - are classified as malicious. As a consequence, it is possible that test cases which manage to detect the sandbox could still be classified as malicious, if they do it in a way that triggers a certain behavior on an instruction level. Therefore, the theories that the authors find most likely regarding the mysterious result of Vendor E are that there is low level monitoring in combination with a powerful anti-virus software installed on the images that run in the sandbox.

If one would consider test cases successful only on the premise that they detect the sandbox, i.e. independently of whether or not they are classified as malicious by the sandbox, the test result would look significantly different. Figure 6.1 shows this result, where the *no exec* results are considered successful instead of unsuccessful, and as previously stated there is a clear difference primarily in the result of Vendor E compared to the official test result in Figure 5.8. Since the authors defined the criteria for successful and unsuccessful test cases themselves, it is possible that others - doing the same experiment - would consider test cases successful on different criteria, e.g. those which detected the sandbox independently of whether or not they were classified as malicious by the sandbox. Comparing the two figures raises an awareness regarding the impact of choosing a certain test success criteria, as well as the impact of the anti-virus of Vendor E (which was the reason behind the majority of the *no exec* cases).

What is surprising in the case of Vendor E is that the code emulation tool works so well in comparison to the sandbox, considering its limited resources. After all, the code emulation tool does not boot an entire operating system like the sandbox does, and thus imitates a "real" environment much worse than the sandbox does. On the other hand, all test cases are designed to detect and bypass

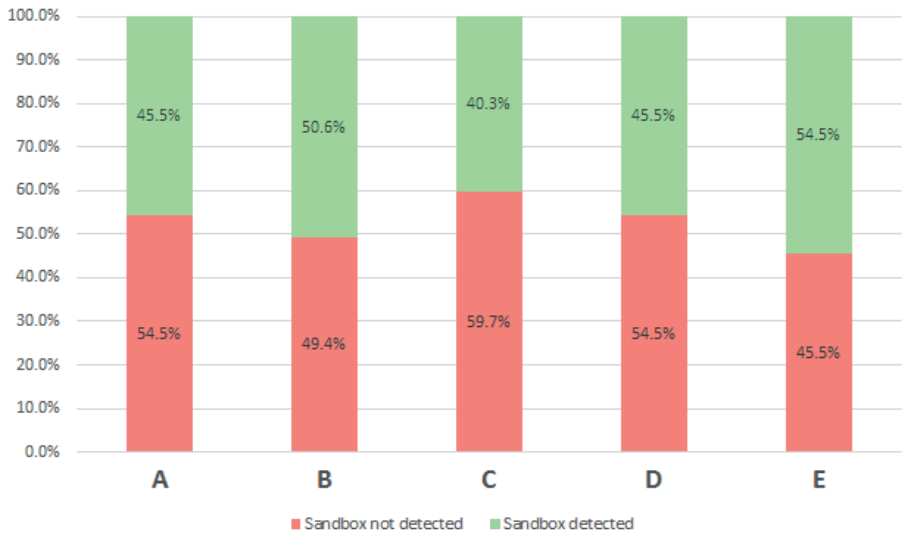


Figure 6.1: Sandbox detection rate per vendor where the test cases detected the sandbox

sandboxes and not code emulation tools; it is possible that the authors, if aware of the fact that code emulation tools were used, could have taken (potentially simple) measures to detect and avoid emulation tools as well.

Except for the results of Vendor E, there were some other interesting findings as well. In the environment test category, Vendor C had the best result thanks to their sandbox supporting the use of golden images. By increasing the screen resolution and changing the keyboard layout, Vendor C detected two more test cases than Vendor E in this category; without golden images, the result would have been a draw between the two.

Speaking of golden images, they were a bit of a disappointment and not as "golden" as one could hope for. First of all, when configuring the golden image of Vendor C, many settings turned out not to be configurable since the sandbox forced a certain value. For instance, in the environment category, the user name was forced to be one which there was already a test case looking for, meaning that although a golden image could easily have detected this test case by setting a "normal" user name, the test case still passed undetected. Furthermore, when studying the results of Vendor B, it turned out that some test cases had passed undetected although they should have been detected with respect to how the golden image was configured. In this case, the sandbox must have forced some of the settings to certain values after the golden image configuration was deployed without giving the person doing the configuration any notification about this. Hence, the way golden images were configured and deployed left quite much to be desired.

Another interesting aspect is networking and how it is configured on the sandboxes. While some of the sandboxes are allowed to, at least partially, communicate over Internet and send e.g. DNS queries, others are completely isolated

and get no Internet access what so ever. The test cases in the network category are designed such that they assume to be on a sandbox if communication fails, if they for instance do not get responses to DNS queries or HTTP requests. This means that isolated sandboxes will do worse in comparison to sandboxes that allow external communication. This fact is the reason for the high coefficient of variation in the network category (35%), and by studying the test results it is easy to tell which sandboxes are most probably isolated (Vendor B, Vendor D) and which ones are not (Vendor A, Vendor C and Vendor E).

6.2 Weakness patterns based on detection technique

Studying the overall result and the malware detection rate per category specifically, it appears that four categories lie slightly below or just around 50% detection rate while the three others are a bit higher and span from 64 to 74%, see Figure 5.9. The four categories with the lower result all lie within a 5% range, and due to the somewhat limited statistical data the number of test cases that actually differ between these categories are very few. Therefore, since the result of these four are almost the same, it is interesting to look at the coefficient of variation as this demonstrates how much the results of the different sandboxes differ from one another.

Where the coefficient of variation is low, there is little variation in the results. In other words, the categories with low detection rate and low coefficient of variation are categories where most sandboxes have low results, meaning that these categories should be seen as the most distinct weakness patterns. Among the four categories with the lowest result, the hardware category has the lowest coefficient of variation, being only 8%, followed by the interactive category with 22%.

It also deserves to be mentioned that the categories with low detection rate and high coefficient of variation will include the real low-water marks. In the timing and network categories, where the coefficient of variation is between 33 and 35%, the minimum detection rates are 30 and 23% respectively. Although these results are very low, they do not constitute a pattern in the same way as the categories with low coefficient of variation.

It might be a bit surprising that the hardware category is where the most distinct weakness pattern is found, since the hypervisor has full control of what hardware its guests believe they have access to. For instance, a hypervisor could easily trick its guests to believe that they have access to more disk than they actually have, and this would not become an issue until the guests fills the entire disk space. In the case of sandboxes - which time out after a few minutes - this disk space would never become a problem in practice, which goes for most other hardware as well: the guests will rarely be able to detect that the reported hardware is not authentic. Therefore, the authors find it remarkable that the sandboxes have not been more successful fooling the virtual machines that they have other hardware than they actually do, since they have every possibility of doing so. Presumably, doing so would not require too much of an effort, and could dramatically improve the result of the hardware category.

The weakness patterns obviously correlate to the efficiency of detection tech-

niques: the most efficient detection techniques are those where the sandboxes have lowest detection rates according to above. However, when switching focus from sandbox weaknesses to test case efficiency, there is another interesting aspect to look at as well, namely the test cases that passed all sandboxes: the *silver bullets*. There are a total of eleven silver bullet test cases which are distributed more or less equally between all but the files category, which has none. Again, due to the limited statistical data, no conclusions will be drawn regarding which categories contain more or less silver bullets than the others. Furthermore, there seems to be no clear pattern among the silver bullets since they are so spread across the different categories.

6.3 Simplicity contra efficiency in detection techniques

Analyzing all test cases on the same premise might be misleading, since some may be more sophisticated than others. Because of this, the authors found it necessary to assess all test cases and rate them based not only on their sandbox detection rate but on their (relative) complexity to implement as well. All test cases were given a simplicity score between one and three, one being harder to implement (time consumption up to one work day) and three being easier (time consumption no more than two hours).

Except for the simplicity to implement, the efficiency of the test case (i.e. its result) was considered as well. The test cases were given two points for each sandbox evaded, resulting in anything between 0 - 10 points, and the total score for each test case was given by multiplying the implementation simplicity score with the efficiency score as follows:

$$\text{Score} = (2 \times \alpha) \times \beta, \quad \alpha = \text{Number of sandboxes evaded}, \quad \beta = \text{Implementation simplicity} \quad (6.1)$$

Using this formula, test cases were given a score ranging from 0 to 30, see Appendix A. A high score means an efficient test case (in terms of detection rate) with low implementation difficulty.

Lastly, for each test case category, the average value of the simplicity-efficiency score was calculated. The scores are shown in Figure 6.2.

The results are surprisingly even across the categories, as it appears that all but one category score between 10-12.¹ On a scale from 0 - 30, the differences between 10 and 12 are negligible which means that all but the files category have very similar relations between simplicity to implement and sandbox detection efficiency. In other words, no specific category is preferable to someone who aims to develop sandbox detection techniques as efficiently as possible.

What might be worth emphasizing in this context is that all test cases consist of only one detection technique each, for the purpose of being able to comparing

¹The files category - which scores just below four - barely contains six test cases, which is roughly half as many test cases as the other categories. This partly undermines its result, as six test cases is somewhat statistically insignificant.

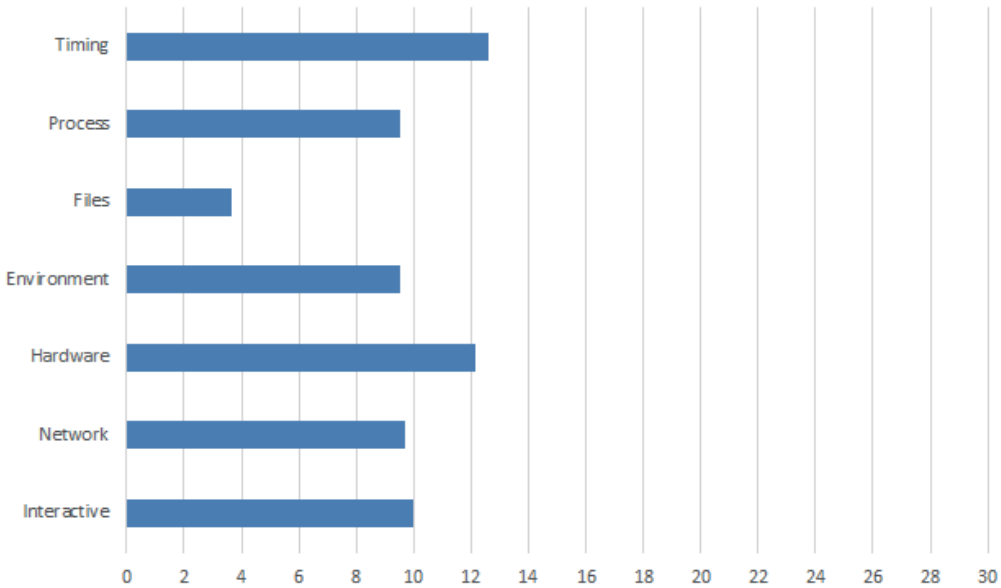


Figure 6.2: Simplicity contra efficiency score per category.

them to each other. In other words, there is no combination of detection techniques, in contrast to what is common in real malware. Combining techniques gives malware greater confidence in concluding whether or not it is executed on a sandbox. If this approach was to be applied on a set of test cases developed in this Master's Thesis, malware with very strong sandbox evasion potential could be developed.

6.4 System lifetime in relation to test results

When attempting to answer the question regarding whether or not the lifetime of the systems (sandboxes) had an impact on the test results, it became obvious that the question was too vaguely defined. First of all, finding the exact age of the products turned out to be significantly harder than expected; although press releases, articles and documentation from all vendors were carefully examined, only indications of the products' lifetimes were found. Second, many of the vendors had bought other vendors over the years, some of which had already been involved in sandboxing for various amounts of time; what would then be the most fair measure of the lifetime of a sandbox? Would it refer to the current product, which was tested in this Master's Thesis, or their very first sandbox? It was decided that the most disambiguous meaning would be the lifetime of the sandboxing technology for that vendor *or* any other vendor acquired by that vendor, since this would indicate how much time the vendor had have to develop their sandboxing technology to what it is today. This definition also made it easier to determine an approximate lifetime for each vendor. These lifetimes are stated in

sections 3.2.1 - 3.2.5.

Three of the vendors - Vendor A, Vendor C and Vendor E- are quite new within the sandboxing business compared to the two others - Vendor B and Vendor D- who have been doing sandboxing for roughly three times as long or more. However, when studying this data in connection to the overall test results, there seems to be no correlation between a long lifetime within sandboxing and good test results, but rather the other way around: Vendor B and Vendor D, both having long experience of sandboxing, are the vendors with the lowest overall test results. Due to the limited statistical data, which includes only five vendors, the authors will not attempt to judge whether this is simply a coincidence or an actual pattern. However, regarding the question of this Master's Thesis concerning whether or not the lifetime of a system has a *significant* impact on its ability to withstand test cases, the answer is "no".

6.5 General discussion and sources of error

As the results have shown, there are categories where there are small differences in the results of the different vendors while there are others where the differences are more significant. The average detection rate of all sandboxes is 56.6%, but only 14% of all test cases are silver bullets. In other words, 86% of all test cases were detected by at least one of the sandboxes - a quite remarkable difference in comparison to the average result of the vendors. This means that by combining several sandboxes, one could achieve a considerably better result than by having only one.

One of the main purposes of this Master's Thesis was to provide Coresec with a decision basis for choosing the most appropriate sandbox on behalf of their customers. While this still holds true, there turned out to be several other aspects to take into consideration as work progressed, which may be considered just as important as the ability of withstanding sandbox detection techniques. This includes:

- the rate of **false positives** among analyzed files. High amounts of false positives will cause frustration as it requires interaction with the configuration of the sandbox, which requires both time and the adequate competence.
- the ease of **setup**, which ranges from minutes to days between the different vendors and might be a priority to less technically oriented customers.
- the ease of **configuration and maintenance**, similar to above.
- the **pricing**, which differs substantially between the different sandboxes but despite this does not necessarily seem to be directly related to the detection rate.

The importance of complementing the test results with an investigation of the aspects above could be illustrated with an over explicit example regarding false positives. In the experiment carried out in this Master's Thesis, a sandbox which classified every file as malicious would be the top achiever since it would

"detect" all malware. Although this was not the case (since all sandboxes failed on a number of test cases), it illustrates the importance of complementing the sandbox detection rate results with experiments concerning other aspects, such as those above.

During the process, Coresec expressed their interest in knowing what rate of "normal" hosts that each test case would execute on in average. Similar to the simplicity score, e.g. an *impact score* could be given each test case based on how many hosts the test case would actually execute on. This impact score would be very interesting, since test cases that manage to detect and bypass sandboxes are of practically no value if they do not also execute on a vast majority of the average hosts; creating test cases that executes on neither of them is trivial. Although the impact score would be highly relevant, a major issue with this approach - which was also the reason why it was never taken - is that it becomes either purely speculative or very time consuming. Making educated guesses regarding how many hosts the test case would execute on and make a scoring accordingly would be of little scientific value. The alternative, quantitative approach of distributing the test cases (without the malware resource, obviously) and analyze the results would be far too time consuming. Consequently, no impact score was given. However, as mentioned in section 1.4, all test cases were tested thoroughly on numerous computers and virtual machines with different configurations.

The detection techniques could be further improved if they were to target the sandbox of a specific vendor. In this case, information about the sandbox could be utilized to customize test cases, something that was never done in this Master's Thesis for the purpose of treating all vendors equally. In reality, it is plausible that a malware developer may focus on just one of the vendors, and may therefore know what defense mechanisms that the target has. Furthermore, some sandbox vendors list their customers publicly on their web pages, and malware developers attacking these customers would have a good idea about the security systems protecting them.

The single biggest source of error in the experiment was, for several reasons, the configuration of the sandboxes. The configuration is critical as it has direct effect on the test result. For instance, some sandboxes had the opportunity of either enabling or disabling Internet access for the virtual machines, while others had support for golden images. Basically, it all came down to a trade-off between equity and utilizing the full potential of the sandboxes: since only some of the sandboxes supported different features, disabling a certain feature for all sandboxes would make the sandboxes as similar to each other as possible. On the other hand, this would also mean that the sandboxes would not reach their full potential, and the test result would become somewhat misleading as the configurations would differ significantly from how the sandboxes were used in practice, where all features can be assumed to be activated. The authors decided to stick to the latter as the guiding principle when configuring the sandboxes, i.e. allowing all sandboxes to be configured according to best effort to utilize their full potential, although this meant that their configurations were quite different from one another. The only exception to this principle was how additional tools were handled: more or less all sandboxes are normally part of a "tool suite" where e.g. firewalls and other tools are also present, and if the sandbox could be configured

to "forward" its samples to other tools in the suite this feature was, if possible, disabled in order to isolate the sandbox and not analyze any other tools.

What is noteworthy, from a user perspective, is that sandbox detection - at least the techniques used in this Master's Thesis - partly is based on virtual machine detection. The hardware category is more or less entirely targeted at detecting virtual machines, and both the process and environment categories include virtual machine detection as well. This means that users running virtual machines would be immune to a substantial amount of the detection techniques used in this Master's Thesis, which very well could be used in real malware as well.

The systems from which the test cases should distinguish sandboxes, referred to as "hosts", are user clients, i.e. the workstations of regular users. Although less common, malware exists which target servers instead of clients. Since servers typically have different configurations compared to clients, regarding both hardware and software, the result of some test cases might be misleading since they have not taken this fact into consideration. For instance, sandboxes may choose not to disguise their server-like hardware, as they aim to resemble both a client and a server at the same time. When running a test case which e.g. looks for server hardware on such a sandbox, the test case will "detect" the sandbox based on something that was actually intended by the vendor of the sandbox and not a misconfiguration. This was pointed out by one of the vendors, whose result was partly affected by this. Although the approach of solving two problems at once by creating a "hybrid" sandbox image might be somewhat problematic, it explains the result of at least one of the vendors. However, one could argue that a better way of solving the problem would be running the analyses in the sandboxes on multiple images with different configurations, e.g. one client and one server image, which was also suggested to the vendor by the authors. The essence of all this is basically that sandboxes should be configured to resemble the environment they protect to the greatest extent possible; if there are a substantial amount of servers in the network protected by the sandbox, having a server-like sandbox image is recommended.

Another interesting aspect, mainly from a vendor perspective, is the level of difficulty of improving the result of the different test case categories. While some categories are difficult and require actions to be taken by the sandbox during the execution of the malware, others are very simple in comparison and could be significantly improved in a matter of minutes. The timing and interactive categories belong to the former of these two, where the sandbox must either somehow speed up execution or simulate user interaction in a realistic way without disclosing this behavior to the file being analyzed, something that is everything but trivial and would require quite an effort both to figure out a solution to and then to implement. In the file and environment categories on the other hand, several of the sandboxes could improve their results significantly simply by configuring the image used. For instance, by doing a couple of minutes of Internet browsing, installing a couple of programs, changing the user name, increasing the screen resolution, adding a hard disk partition and setting a password for the user account, the majority of the sandboxes could raise their detection rate by several per cent without too much effort.

Although extracting information about the sandboxes was difficult, some patterns could be observed in the results based on the technologies used in the tested systems. The sandbox of Vendor E, which has an advanced low level monitoring technique and possibly also a potent anti-virus software installed on its virtual machine images, achieved the best overall results thanks to this. Furthermore, the systems that supported the use of so called golden images could prevent detection techniques that otherwise would have been successful. Lastly, whether the sandboxes allowed its virtual machines Internet access or not had a significant impact on the outcome of the test cases related to this.

The *hardware* and *interactive* test case categories were particularly weak in comparison the the others, in the sense that their detection rates were consistently low. In other words, test cases utilizing detection techniques bases on either of these two categories had a high rate of success on most sandboxes. Although the real low-water marks were found in the network and timing categories, some of the sandboxes also did well in these categories, why they are not to be regarded as patterns in the same way as hardware and interactive.

By grading all test cases on a scale from one to three and correlating this score to the test results, there turned out to exist test cases that were both simple to develop and very efficient at detecting sandboxes.

The most efficient sandbox detection techniques are directly related to the patterns of weaknesses in the sandboxes, which means that the hardware and interactive based test cases are to be seen as most efficient. Whether this efficiency is *significantly* higher than that of the other categories is left to the reader to judge, see Figure 5.9. The "silver bullet" test cases, i.e. the test cases that succeeded and passed undetected on all sandboxes, were somewhat evenly distributed across all categories and did not constitute a clear pattern.

The lifetime of a system, which was defined as the amount of time for which the vendor of each system had been involved in the sandboxing industry, did not have a significant impact on the systems' ability to withstand attacks. Surprisingly, it was rather the other way around: the test results suggested that vendors who got into the sandboxing business more recently actually achieved better results in general.

The authors find it surprising that sandbox detection proved not to be harder than it turned out, as many of the test cases which were trivial to develop actually succeeded on several of the sandboxes. They are therefore convinced that all

vendors could improve their test results significantly without too much effort, simply by configuring the image(s) of the sandboxes properly. The authors have provided all vendors with the test results and the test cases, in order to help them fighting malware which deploy sandbox detection techniques.

This Master's Thesis has proven that bypassing sandboxes can be done in a number of ways. However, it should be mentioned that the sandbox is normally only one component in a suite of other security tools, and that these must be bypassed as well for a successful infection.

7.1 Future work

This Master's Thesis has focused solely on sandbox detection techniques using executable files. Since malware also appear in for example PDF files and Microsoft Office documents, it would be interesting to do a similar research using these file types instead; this could possibly open up for new detection techniques.

The sandbox detection techniques developed in this Thesis do not form an exhaustive list of techniques for this purpose; they are merely a result of the information gathering and ideas of the authors. An area of techniques that the authors believe to be very efficient, which unfortunately fell out of scope due to time constraints, are test cases written in assembly. Because of the way hooking is implemented in the sandboxes, the authors suspect that assembly test cases could be considerably more effective.

An interesting aspect is the way the sandboxes get access to files to analyze when the traffic reaching them is encrypted. Today an increasing amount of traffic is encrypted using HTTPS, and the sandboxes are completely blind to this traffic unless they are bundled with another tool (a proxy of some kind) which is capable of performing HTTPS decryption [47]. In other words, for inline sandboxes to work well in practice, they depend heavily on a good HTTPS decryption tool, which is another aspect not taken into consideration in this Master's Thesis. Apparently, the different vendors have proxies of very varying quality - some of them even lack this completely - which would very much affect the result of the sandboxes when used in practice in a real environment. Again, although this would be highly interesting to dig deeper into, comparing proxies of the different vendors unfortunately fell out of scope of this Master's Thesis, but could be subject to future work.

Finally, an area which has gotten a lot of attention recently and continuously evolves is machine learning. Machine learning is relevant within many IT related fields, and within the computer security field it has created a new way of analysing malware. By making machines capable of learning themselves the difference between benign and malicious files, there is a chance of seeing a dramatic improvement in malware detection capabilities [48]. The method is still on the rise and needs further exploration, why experiments on the topic would be highly interesting.

References

- [1] A. Mesbahi and A. Swinnen, "One packer to rule them all: Empirical identification, comparison and circumvention of current antivirus detection techniques," in *Black Hat, USA*, 2014.
- [2] AV-TEST Institute, "Malware statistics." available at: <https://www.av-test.org/en/statistics/malware/>. Last accessed: 2015-04-12.
- [3] Symantec, "Internet security threat report 20." https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf. Last accessed: 2015-02-01.
- [4] S. S. Hansen, T. M. T. Larsen, M. Stevanovic, and J. M. Pedersen, "An approach for detection and family classification of malware based on behavioral analysis," in *2016 International Conference on Computing, Networking and Communications (ICNC)*, pp. 1–5, Feb 2016.
- [5] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. San Francisco, CA: No Starch Press, fifth printing ed., 2012.
- [6] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, "Semantics-based online malware detection: Towards efficient real-time protection against malware," *Information Forensics and Security, IEEE Transactions on*, vol. 11, pp. 289–302, Feb 2016.
- [7] Cuckoo Foundation, "Cuckoo sandbox v. 2.0 rc1." <https://www.cuckoosandbox.org/>.
- [8] S. Mohd Shaid and M. Maarof, "Malware behavior image for malware variant identification," in *Biometrics and Security Technologies (ISBAST), 2014 International Symposium on*, pp. 238–243, Aug 2014.
- [9] H. Dornhackl, K. Kadletz, R. Luh, and P. Tavolato, "Defining malicious behavior," in *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, pp. 273–278, Sept 2014.
- [10] Kaspersky Lab, "Who creates malware?." available at: <https://usa.kaspersky.com/internet-security-center/threats/who-creates-malware>. Last accessed: 2015-04-15.

- [11] S. Cobb and A. Lee, "Malware is called malicious for a reason: The risks of weaponizing code," in *Cyber Conflict (CyCon 2014)*, 2014 6th International Conference On, pp. 71–84, June 2014.
- [12] C. Czosseck, G. Klein, and F. Leder, "On the arms race around botnets - setting up and taking down botnets," in *2011 3rd International Conference on Cyber Conflict*, pp. 1–14, June 2011.
- [13] Netmarketshare, "Desktop operating system market share." available at: <https://www.netmarketshare.com/operating-system-market-share.aspx>. Last accessed: 2015-04-15.
- [14] A. Javed and M. Akhlaq, "Patterns in malware designed for data espionage and backdoor creation," in *Applied Sciences and Technology (IBCAST)*, 2015 12th International Bhurban Conference on, pp. 338–342, Jan 2015.
- [15] E. Alomari, S. Manickam, B. B. Gupta, P. Singh, and M. Anbar, "Design, deployment and use of http-based botnet (hbb) testbed," in *16th International Conference on Advanced Communication Technology*, pp. 1265–1269, Feb 2014.
- [16] W. Peng, G. Qingping, S. Huijuan, and T. Xiaoyi, "A guess to detect the downloader-like programs," in *Distributed Computing and Applications to Business Engineering and Science (DCABES)*, 2010 Ninth International Symposium on, pp. 458–461, Aug 2010.
- [17] W. Tsaur, "Strengthening digital rights management using a new driver-hidden rootkit," *Consumer Electronics, IEEE Transactions on*, vol. 58, pp. 479–483, May 2012.
- [18] L. Hu, T. Li, N. Xie, and J. Hu, "False positive elimination in intrusion detection based on clustering," in *Fuzzy Systems and Knowledge Discovery (FSKD)*, 2015 12th International Conference on, pp. 519–523, Aug 2015.
- [19] M. Ramilli and M. Prandini, "Always the same, never the same," *IEEE Security Privacy*, vol. 8, pp. 73–75, March 2010.
- [20] S. Ravi, N. Balakrishnan, and B. Venkatesh, "Behavior-based malware analysis using profile hidden markov models," in *Security and Cryptography (SECRYPT)*, 2013 International Conference on, pp. 1–12, July 2013.
- [21] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pp. 421–430, Dec 2007.
- [22] B.-D. Yoon and O. N. Garcia, "Cognitive activities and support in debugging," in *Human Interaction with Complex Systems, 1998. Proceedings., Fourth Annual Symposium on*, pp. 160–169, Mar 1998.
- [23] M. Zolotukhin and T. Hamalainen, "Detection of zero-day malware based on the analysis of opcode sequences," in *Consumer Communications and Networking Conference (CCNC)*, 2014 IEEE 11th, pp. 386–391, Jan 2014.

- [24] K. Yoshioka, Y. Hosobuchi, T. Orii, and T. Matsumoto, "Vulnerability in public malware sandbox analysis systems," in *Applications and the Internet (SAINT), 2010 10th IEEE/IPSJ International Symposium on*, pp. 265–268, July 2010.
- [25] A. Albertini, "Portable executable 101, version 1." <http://corkami.googlecode.com/files/PE101-v1.pdf>. Last accessed: 2015-01-28.
- [26] Microsoft, "Microsoft portable executable and common object file format specification, revision 8.3. 2013." available at: http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/pecoff_v83.docx. Last accessed: 2015-02-15.
- [27] Microsoft, "Resource types." available at: [msdn.microsoft.com/en-us/library/windows/desktop/ms648009\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms648009(v=vs.85).aspx). Last accessed: 2015-02-15.
- [28] M. Franz, "Dynamic linking of software components," *Computer*, vol. 30, pp. 74–81, Mar 1997.
- [29] Microsoft, "Dynamic-link libraries." available at: [https://msdn.microsoft.com/en-us/library/windows/desktop/ms682589\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682589(v=vs.85).aspx). Last accessed: 2015-03-31.
- [30] J. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, pp. 32–38, May 2005.
- [31] Z. Xiao, W. Song, and Q. Chen, "Dynamic resource allocation using virtual machines for cloud computing environment," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, pp. 1107–1117, June 2013.
- [32] VMware, "Using snapshots to manage virtual machines." available at: http://pubs.vmware.com/vsphere-60/topic/com.vmware.vsphere.vm_admin.doc/GUID-CA948C69-7F58-4519-AEB1-739545EA94E5.html. Last accessed: 2015-02-16.
- [33] Sandboxie Holdings, "How it works." available at: <http://www.sandboxie.com/index.php?HowItWorks>. Last accessed: 2015-02-16.
- [34] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using cwsandbox," *Security Privacy, IEEE*, vol. 5, pp. 32–39, March 2007.
- [35] VMware, "Vmware tools components." available at: https://pubs.vmware.com/vsphere-4-esx-vcenter/index.jsp?topic=/com.vmware.vsphere.vmadmin.doc_41/vsp_vm_guide/installing_and_upgrading_vmware_tools/c_vmware_tools_components.html. Last accessed: 2015-05-17.
- [36] B. Mariani, "Inline hooking in windows." available at: https://www.htbridge.com/blog/inline_hooking_in_windows.html. Last accessed: 2015-03-29.

- [37] B. Bill, *The Rootkit Arsenal*. Burlington, VT: Jones & Bartlett learning, 2 ed., 2013.
- [38] Microsoft, "User mode and kernel mode." available at: [https://msdn.microsoft.com/en-us/library/windows/hardware/ff554836\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/hardware/ff554836(v=vs.85).aspx). Last accessed: 2015-03-29.
- [39] A. Singh and Z. Bu, "Hot knives through butter: Evading file-based sandboxes," in *Black Hat, USA*, 2013.
- [40] Z. Balazs, "Malware analysis sandbox testing methodology," in *Botconf, France*, 2015.
- [41] Y. Gao, Z. Lu, and Y. Luo, "Survey on malware anti-analysis," in *Intelligent Control and Information Processing (ICICIP), 2014 Fifth International Conference on*, pp. 270–275, Aug 2014.
- [42] M. Vasilescu, L. Gheorghe, and N. Tapus, "Practical malware analysis based on sandboxing," in *2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference*, pp. 1–6, Sept 2014.
- [43] Kaspersky Lab, "What is a trojan virus? - definition." available at: <https://usa.kaspersky.com/internet-security-center/threats/trojans>. Last accessed: 2015-04-19.
- [44] Kaspersky Lab, "What is beta bot? - definition." available at: <http://usa.kaspersky.com/internet-security-center/definitions/beta-bot>. Last accessed: 2015-02-23.
- [45] Symantec, "Trojan.betabot." available at: https://www.symantec.com/security_response/writeup.jsp?docid=2013-022516-2352-99. Last accessed: 2015-02-23.
- [46] Department of Homeland Security, "Daily open source infrastructure report 23 september 2013." <https://www.dhs.gov/sites/default/files/publications/nppd/ip/daily-report/dhs-daily-report-2013-09-23.pdf>. Last accessed: 2015-02-23.
- [47] J. M. Butler, "Finding hidden threats by decrypting ssl," in *SANS Analyst Whitepaper*, 2013.
- [48] I. Firdausi, C. lim, A. Erwin, and A. S. Nugroho, "Analysis of machine learning techniques used in behavior-based malware detection," in *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*, pp. 201–203, Dec 2010.

Test case simplicity-efficiency score

Table A.1: Timing category

#	Simplicity	Efficiency	Total score
1	3	0	0
2	3	4	12
3	2	6	12
4	2	4	8
5	2	6	12
6	2	0	0
7	3	4	12
8	2	10	20
9	3	10	30
10	2	10	20

Table A.2: Process category

#	Simplicity	Efficiency	Total score
1	2	0	0
2	2	4	8
3	2	6	12
4	1	2	2
5	3	4	12
6	3	0	0
7	3	0	0
8	2	10	20
9	2	0	0
10	3	0	0

Table A.3: Files category

#	Simplicity	Efficiency	Total score
1	2	0	0
2	1	4	4
3	2	2	4
4	3	0	0
5	2	4	8
6	1	6	6

Table A.4: Environment category

#	Simplicity	Efficiency	Total score
1	3	2	6
2	3	10	30
3	3	8	24
4	3	6	18
5	3	0	0
6	2	6	12
7	2	6	12
8	2	4	8
9	3	0	0
10	3	8	24
11	1	0	0
12	3	2	6
13	2	6	12
14	2	0	0
15	2	0	0
16	2	0	0

Table A.5: Hardware category

#	Simplicity	Efficiency	Total score
1	2	8	16
2	3	4	12
3	3	6	18
4	1	4	4
5	1	6	6
6	3	6	18
7	2	2	4
8	2	0	0
9	2	10	20
10	2	0	0
11	3	8	24
12	3	2	6
13	3	10	30

Table A.6: Network category

#	Simplicity	Efficiency	Total score
1	2	10	20
2	2	0	0
3	1	2	2
4	2	6	12
5	1	6	6
6	1	8	8
7	2	6	12
8	3	0	0
9	3	8	24
10	1	0	0
11	2	10	20
12	2	8	16
13	1	6	6

Table A.7: Interactive category

#	Simplicity	Efficiency	Total score
1	3	0	0
2	3	2	6
3	2	0	0
4	2	6	12
5	1	8	8
6	3	10	30
7	3	8	24
8	1	10	10
9	3	0	0



LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2016-517

<http://www.eit.lth.se>