

Automated testing of antivirus coverage

Kris-Mikael Krister
`krismika-at-stud.ntnu.no`

TDT4560 - Safety and Security in IT Systems
Torbjørn Skramstad ← main supervisor

Fall 2008

Contents

List of Figures	iii
List of Tables	v
Listings	vi
1 Introduction	5
1.1 Problem description	5
1.2 Result goals	6
1.3 Terminology and acronyms	7
1.4 Methodology	9
1.5 Document structure	9
2 Problem domain and project rationale	11
2.1 Factors that facilitate the malware spread	11
2.2 Countermeasuring malware	12
2.3 A brief introduction to NAAS	17
3 Malware analysis	19
3.1 The analysis process	19
3.2 Analysis methods	21
3.3 Surface analysis tools	23
3.4 Dynamic analysis tools	23
3.5 Static analysis tools	25
4 Towards a solution	29
4.1 Existing related work and possible solutions	29
4.2 High level requirements for a new solution	32
4.3 Platform virtualization	33
4.4 Virtualization software	39
4.5 Differences in the VMware software packages	40
4.6 The VMware Programming API (VIX)	41
5 Requirements specification	43
5.1 Use case analysis	43
5.2 Some use case scenarios	49
5.3 Overall list of requirements	54

5.4	Mapping requirements	56
6	Design	59
6.1	Description of components	59
6.2	Choice of programming languages	61
6.3	Design of the first part, virtual machine deployment	61
6.4	Design of the second part, the <code>vmcom</code> program	63
6.5	Design of the third part, the NAAS integration module	65
6.6	Design limitations	67
7	Implementation	69
7.1	Implementing the first part, Virtual machine deployment	69
7.2	Implementing the second part, the <code>vmcom</code> program	69
7.3	Implementing the third part, the NAAS integration module	72
7.4	Performance	73
8	Project evaluation	77
8.1	Fulfilment of project goals and system requirements	77
8.2	Involved parties	78
8.3	Further work	80
8.4	Challenges	82
9	Project conclusion	85
	References	87
A	User manual	91
A.1	Setting up and configuring the system	91
A.2	Available command line switches	93
B	<code>vmcom</code> in detail	97
B.1	Complete source code	97
B.2	On demand arguments and scripts	116
B.3	Activity diagram	119
C	Tool screenshots	121

List of Figures

2.1	Increase in the amount of discovered software vulnerabilities during the time span from 1997 to 2007.	12
2.2	Response times of antivirus product vendors on major malware outbreaks.	14
3.1	Graphical representation of the different steps of a malware analysis, including antivirus coverage testing which is the task for this project to arrange.	22
4.1	Screenshot of a Microsoft Windows XP operating system running from an Ubuntu operating system installation.	34
4.2	Graphical display of the different protection rings.	35
4.3	Architecture diagram of a computer with one operating system. .	36
4.4	Architecture diagram of a computer with one main operating system with two virtual guest operating systems installed on top. .	37
5.1	Overall use case diagram.	44
5.2	Use case diagram for “Scan sample”.	45
5.3	Use case diagram for “Revert to clean states”.	47
5.4	Use case diagram for “Update virus definitions”.	48
5.5	Sequence diagram showing interactions for the three first scenarios. .	52
5.6	Sequence diagram showing interactions for the last scenario. . . .	53
6.1	Overview of system parts and their interactions in the deployment diagram notation.	62
6.2	The NAAS integration displays how different third party “tools” can connect to the main NAAS system.	66
7.1	Running times for each operation on the virtual machine.	74
7.2	Total utilisation of CPU during scanning of a file.	75
7.3	Total utilisation of system memory during scanning of a file. . . .	76
B.1	Activity diagram showing vmcom program flow.	120
C.1	Screenshot showing usage of tools for three of the most common hash sum methods used during the surface analysis phase.	121

C.2	Screenshot showing usage of the network analysis tool, TCPView .	122
C.3	Screenshot showing usage of the network analysis tool, Wireshark .	123
C.4	Screenshot showing usage of the static analysis tool, IDA Pro .	124
C.5	Screenshot showing usage of vmcom .	125

List of Tables

1.1	An overview of terms and acronyms in the report.	8
2.1	Scan result showing false positives by 10 of 36 total antivirus products used.	16
3.1	An overview over the analysis tools described.	27
4.1	Conflicts between antivirus products on the same operating system.	31
4.2	Important differences in the two VMware virtualization products of current interest.	41
4.3	Functionality needed in the design and the respective VIX function calls available to use.	42
5.1	Use case for “Scan sample”.	45
5.2	Use case for “Submit sample through NAAS”.	46
5.3	Use case for “Revert to clean states”.	47
5.4	Use case for “Update virus definitions”.	48
5.5	Requirements for the implementation part of the project.	56
6.1	On-demand features in, and cost of a list of different antivirus products.	68
7.1	Antivirus products and their on demand features supported by the implemented solution.	70
8.1	Summarised list of the accomplishment of result goals and stated functional and non-functional requirements.	79

Listings

4.1	C code returning a non-zero value if a virtualized environment is detected. Only a few lines of carefully selected code is necessary for a detection.	39
6.1	Pseudocode for on demand automatic scanning of one sample file.	64
7.1	Code example showing subroutine for copying a file from the host operating system to a guest virtual machine.	71
7.2	Example parser code for one of the antivirus products.	71
7.3	Python example code showing the necessary code to integrate <code>vmcom</code> into the NAAS framework.	72
A.1	Perl code template for adding and enabling a virtual machine.	93
B.1	Complete source code for the <code>vmcom</code> program.	97

Summary

Threatening software is located all over the Internet, often hidden or obfuscated to camouflage dangerous content. Deducing the real functionality from such software is a tedious and time consuming process, and job automation should be applied wherever possible. Antivirus products might be used to decrease the amount of time required to analyse the software, but more than just one antivirus product is often needed to get any usable results. This report covers a study of utilising a set of multiple antivirus products to scan the same data for viruses in an automated manner. Additionally, a system able to perform virus scans by utilising an arbitrary amount of antivirus products concurrently is implemented.

Preface

This report is the result of a problem stated by the Norwegian Computer Emergency Response Team (NorCERT), a division under the Norwegian National Security Authority (NSM). The task was structured and written by Kris-Mikael Krister, attending the Norwegian University of Science and Technology (NTNU), Department of Computer and Information Science (IDI) and was accomplished autumn 2008. I would like to thank my internal supervisor, Torbjørn Skramstad from IDI NTNU for assistance and guidelines regarding content of the report, and my external supervisor Lars Haukli from NorCERT for making this project possible.

Oslo, December 2008



Kris-Mikael Krister

Chapter 1

Introduction

The amount of suspicious software called viruses, spyware and malware being spread on the Internet increases steadily and rapidly [KBS⁺07]. The suspicious software forms a threat to the Internet users and should be avoided to sustain secure computer environments. Solutions able to countermeasure the threats are therefore needed. One of the methods to stop such malicious software is using programs that recognise known threats. These programs are commonly called *antivirus applications*. An antivirus application must be fed with instructions, a special signature, for each potential threat to be able to recognise it. The antivirus vendors must therefore release signature updates regularly, but the current increase in discovered suspicious software makes it hard to keep up with. There exists numerous different antivirus applications, each more or less with its own database of known threats. If there is some way to use a multiple of the antivirus applications when trying to recognise the threat, the chance of deducing any malicious functionality increases. If at the same time this process can be automated, a lot of time can be saved from the people working with malware.

This project will, in cooperation with NorCERT, venture through the process of finding the best way of managing an automated detection by the use of multiple antivirus products. In addition, a software implementation of such a system will be developed and released freely available for download and use.

1.1 Problem description

The following text concretise the problems to be addressed in this project. It is written in cooperation with representatives from NorCERT, and approved by the internal supervisor.

The increasing number of malware requires an analyst to more quickly gain information about suspicious samples of software. However, analysing malware is challenging and often time consuming. Anti-virus products can be used to scan samples and give the analyst

an indication of what kind of malicious code it contains, but newly introduced malware usually have low detection coverage by the different antivirus products. False negatives will be introduced for this reason, but the chance of eliminating these increase by applying more than one antivirus product on the same data. Sadly, it does not scale very well when done manually, and it is cumbersome and often not feasible to accomplish on the same operating system (OS). One way to solve this is by separating the antivirus installations in their own virtual environment and automatically collect and process the results gained from the different installations.

The thesis should include a brief study of methods and tools used in the field of malware analysis under a controlled and secured environment, and in addition to this, the student should:

- 1 Examine the possibility of automated virus scanning on suspected suspicious software. Several antivirus products must be supported, each installed on its own virtual operating system environment. The process should be controlled by a centralized host, which may submit samples and aggregate the results.
- 2 Implement the solution as a module for the NorCERT Artifact Analysis System (NAAS), which will then support the analyst with information of the submitted samples swift and early in the analysis phase.

The assignment can be continued as a master thesis where further improvements in the process of analysis are studied.

1.2 Result goals

A few concise goals are stated, each derived from the problem description and labeled for a referring purpose.

RG.01 Deliver a general view of how, and why, analysis of malware is done, including common methods used and a brief toolkit survey.

RG.02 Propose a design of how to achieve automated use of antivirus applications under *virtual environments*.

RG.03 Deliver software that has the functionality from **RG.02** and can be used as a standalone program.

RG.04 Deliver software that has the functionality from **RG.02** and can be used through the NAAS system.

RG.05 Contribute with background material for further projects or studies in the field of malware analysis.

1.3 Terminology and acronyms

There are some terms used in this report that might be interpreted ambiguously and their meaning are therefore explained in this section. For a quick overview including acronyms, see Table 1.1 on the following page.

The *malware* term is used throughout this document to signify a large group of software containing a number of more specialised groups of software like *trojan horse*, *logic bomb*, *worm*, *spyware* and *virus*. Each of these groups has their own definition, but share an important property - they are all software written to compromise the integrity, confidentiality or availability¹ of a victim's data. The literature about malware is wide and sometimes ambiguous. This makes a proper definition of these specialised terms difficult to specify. This report will not directly handle the differences in these groups, but treat all the same under the term “malware”, and use “malicious code” and “malicious software” interchangeably.

The *sample* term is used in this report to describe a limited quantity, usually one file characterised as a suspicious object which is subject for *analysis*. During the analysis process, the functionality and goals of the sample are meant to be deduced. The term *malware analyst*, or the shortened form *analyst*, is used to signify the person performing the actual analysis.

An implemented system for managing antivirus applications will be one of the results from this project. The chapters will refer to this system as the “system” in general, but also the *system-to-be* prior to its realisation.

The antivirus coverage term signifies the ratio between antivirus products that do and do not recognise a malware sample. A low value means that few antivirus products recognise the malware. For a known sample, the antivirus coverage (AC) is shown in Equation (1.1). The utopian value for the antivirus coverage would obviously be “1” for each malware sample. A synonym to antivirus coverage used in this report is “antivirus detection ratio”.

$$AC = \frac{\text{Number of antivirus products that recognise the malware}}{\text{Total amount of antivirus products}} \quad (1.1)$$

Newly introduced non trivial terms are marked as *emphasised text* the first time they are written, and are given a description. How detailed this description is depends upon the importance of the term regarding the content of the document. Command line text and application names are written with a **typewriter font** to make it clear that it is in fact a reference to a program, a command line function or similar.

¹The three terms “integrity”, “confidentiality” and “availability” together form the basis of “software security” [McG06].

Term	Comment
Malware	Signifies a large group of software written to compromise the security of a victim's data.
Malicious code/software	Same as above
Sample	A suspicious file subject to a malware analysis.
Malware analyst	Person performing the malware analysis.
System (to-be)	The result of the implementation part of the project.
Attacker	A person which is performing actions with malicious intentions.
Software vulnerability	A software weakness which may lead to a possible compromise in the security.
Antivirus coverage	For a known malware sample, the antivirus coverage is the ratio between antivirus products that do and do not recognise the malware.
Virus detection ratio	Same as above.
(a) Terminology	
Acronym	Meaning
OS	Operating system
VM	Virtual machine
AV	Antivirus
(b) Acronyms	

Table 1.1: An overview of terms and acronyms in the report.

1.4 Methodology

Preliminary studies in this project are based on various existing literature and information from web pages, forums and online documents. At the study of background information, articles and web pages were the most important elements used as sources as they were considered good enough. References to books and articles are in general used as bibliography only, and do not need to be read throughout.

Parts of the preliminary studies are a survey of state of the art tools and practices used by malware analysts at NorCERT. Not everything is covered in this report due to time constraints, but the elements most frequently used at NorCERT are selected.

In addition to the preliminary studies, a major part of this project is the implementation of a system not only as proof of concept, but to be used in real life situations. The development process used was a traditional waterfall process [Bra00], and is documented and structured in this report accordingly. To avoid unnecessary huge listings of code in the report, only example code is shown. Full source code can be found in the appendices for the readers of special interest.

Reasoning for conclusions and choices made are given where appropriate. Choices that later on proved not to be optimal are discussed in Section 8.3 on page 80.

1.5 Document structure

This report is structured in eight main parts. Chapter 2 discusses problems regarding the use of antivirus products, and continues with Chapter 3, which covers methods and tools used by analysts working with malware. Chapter 4 contains a discussion of how to solve the described problem and which solutions and techniques can be used. The following part, Chapter 5, elicit requirements for the system-to-be, and continues in Chapter 6 with a more detailed description of the components in the system. Chapter 7 lists the implemented solution of the system and explains the most important functions of the produced code. An evaluation of the project in general can be found in Chapter 8, and the report finalises with a conclusion in Chapter 9.

Chapter 2

Problem domain and project rationale

This chapter contains a survey of malware, the common countermeasures against malware and their respective positive and negative sides. The chapter will reason for the project's problem description, and give an overview of the current situation regarding threats from malware. The chapter is split in two main parts, starting with a brief introduction to the threats from malware in Section 2.1. The latter part, Section 2.2, discusses how people stand against the threats, and mentions the problems these approaches have. Additionally, a brief introduction to the artifact analysis system used at NorCERT can be found in Section 2.3.

2.1 Factors that facilitate the malware spread

A goal for malware is often to infect as many clients as possible. In general there are two main areas that malware use to multiply itself. One of them is utilising weaknesses in software code [Hei04], and as the amount of flaws and weaknesses in software code continues to increase every year [KBS⁺07], this tends to get easier to accomplish. Figure 2.1 on the following page shows the amount of vulnerabilities reported annually since 1996, and gives the indication that vulnerabilities in software are not likely to disappear all of a sudden.

There might be several reasons for this, but as software grows in size and tend to get more complex, the available code to attack also increases. The data is collected from the Common Vulnerability Enumeration (CVE), which could be considered the de facto standard source for collecting and storing information about software vulnerabilities. At the first glance, it is easy to believe that these numbers will increase the coming years, but that is not necessarily the case be so. Software developers are getting more focused on producing secure code [vWM05], and the new operating systems are designed to prevent common methods of exploiting vulnerabilities [Ahm07], such as trying to overflow a predefined buffer (buffer overflow) [Sca07].

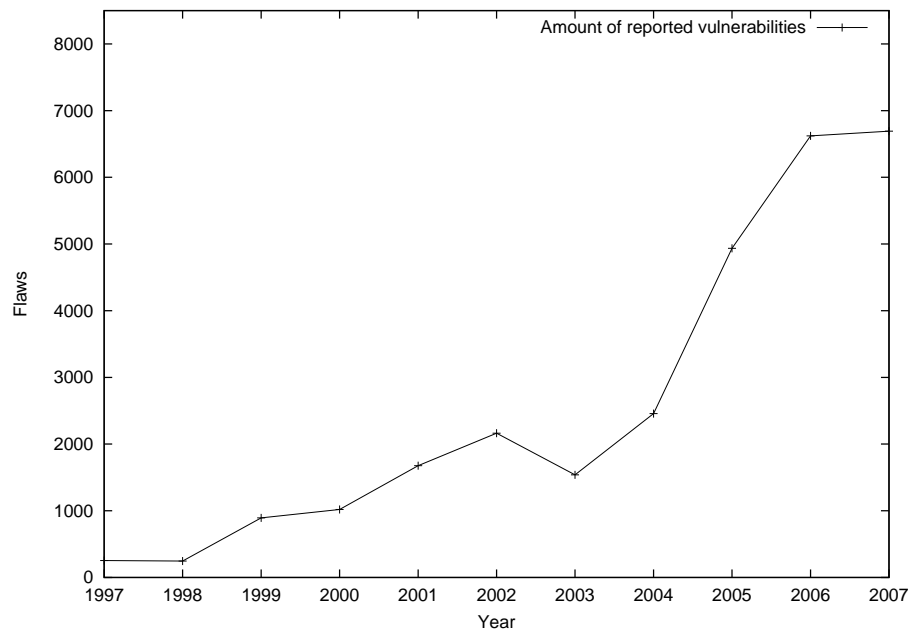


Figure 2.1: Increase in the amount of discovered software vulnerabilities during the time span from 1997 to 2007 [cve08]. The figure is shown to indicate that software vulnerabilities does indeed exist.

The other method malware use to spread itself is by misusing the fact that people are often blissfully ignorant about threats they currently are facing on the Internet. As an example, an increase in malware spread in instant messaging (IM) have been reported over the last years [Lea05]. By using the interfaces exposed by the IM clients, malware can send messages to all contacts an infected user has on his or her list. To be able to infect more machines, the message usually recommends to click on a (malicious) link that installs malware [HC03].

Regularly, the process of infection utilises a combination of the two methods as the IM example just shown shows. The first step would be to trick a user clicking on a link that installs malicious programs. This program could then try to exploit one or more vulnerabilities on the victim's machine. If the attack is successful, it is possible that the attacker takes total remote control over the machine and renders the computer insecure.

2.2 Countermeasuring malware

The average person believes he is safe from malicious code and its threats if an antivirus product is installed on his computer, preferably protected by an active firewall of some kind running in the background. This has been called *unrealistic optimism* and is, sadly, far from the current reality [CGME07]. While a firewall

and an antivirus application are the two most common countermeasures used against malware (at least by the home users), they are not exactly bullet proof nor perfect. Instead of preventing all threats, they merely *assists* the user to avoid some of them.

2.2.1 Antivirus applications

The software programs used to identify and neutralise malicious software are called *antivirus* software, even though the malware do not need to be a virus by definition. Most of the modern antivirus products try to remove all kind of harmful or unwanted programs, not just the viruses. Still, the name clinger from historical reasons when “virus” was the only current software threat, and the name seems to be stuck for at least the two following important reasons.

- The term “antivirus” is well-known by the user community, and a large percentage of users of computer software believe that there is mandatory to include an antivirus product in every computer. Changing the product name to “antimalware” or similar could cause confusion among potential buyers, even though the name is a better description for what the software actually does.
- Using such a broad terminology allow the vendors to sell products with more or less the same functionality with different names. “Antivirus”, “PC-Cleaner” and “antispyware” are all usable product names. Whether or not playing on users lack of knowledge is a fair method of marketing will not be discussed further, but it is important to know this fact is being exploited.

A large number of different vendors are selling antivirus products, and their virus signature databases are increasing every day as new malware are discovered. The products can be utilised by a malware analyst to check if a sample is consider safe or not by the product, and use the result data early in a possibly longer analysis. If the virus scan gives a positive result, the antivirus application considers the scanned software harmful - while a negative result from the scan means a safe file. The analyst can save a lot of time and effort if his or her antivirus product reports the file suspicious with a named threat. The analyst can then use this information to look up the threat’s functionality in (publicly available) information databases without the need of further time consuming analyses. If the product does not recognise the file as harmful, and a deeper analysis finds the file malicious, an indication of how fresh the malware is can be made. Maybe none of the antivirus vendors have ever seen this kind of sample or maybe even it is custom made and targeted against one particular company or person. In either way, important information can be gained by using antivirus products in an analysis. Additionally, antivirus products can be used to make the analysis process swift, but there are some issues to consider;

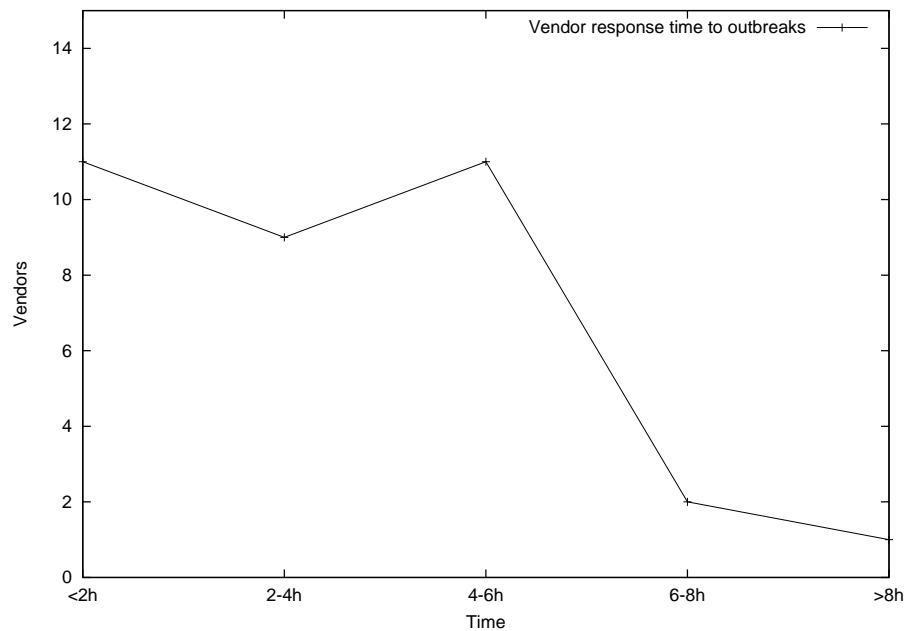


Figure 2.2: Response times of antivirus product vendors on major malware outbreaks [av-08].

False negatives and low detection ratios A negative result does not guarantee that the scanned sample is safe, it is just a consideration from that particular antivirus product, and should be used as an indication only. A *false negative* means a scan gives a negative result where it *should* give a positive result. New malware is introduced daily and the antivirus products often fail to catch them before the vendors have created and published new signatures. The malware will therefore not be detected until the antivirus software has been updated with the newest signatures, and the computers are threatened during this time gap. As malware becomes increasingly intelligent, this becomes even more imminent since the software often tries to avoid the signature based detection methods. Figure 2.2 shows the time span from antivirus vendors responding to *major*¹ malware outbreaks. During this time, the users are completely exposed to the new threats. Additionally, the time before the client actually upgrades his or her product is often significant, and the total time of full exposure increases further.

False positives A *false positive* is the opposite of a false negative. A false positive emerges when the antivirus product considers a safe file harmful. A

¹It emerged in a discussion with senior virus analytic in Norman Data Defence, that “major” in this context covers most malware that is scheduled for detection as a signature, not only the ones that wreak havoc in the society.

positive result can assist the analyst with swift and easy indication of what kind of malware the sample contains. By the use of named virus signatures, the malware can often be looked up on open web sources like virus information databases and its functionality can be learned quickly. If however the result is a false positive, there might be confusion and wrong conclusions are taken. False positives are easily introduced by antivirus software using heuristic scanning techniques [Sch02]. As an example of this, consider the legit mIRC patch “FiSH 1.30”² The patch offers chat encryption to clients which are communicating using IRC networks with the mIRC program. Since the source code for mIRC is closed and the program executable is pre-compiled prior to installation, the patch is required to tamper with the binary code executable to load the requested behaviour into the mIRC program. In addition, the binary is runtime packed and needs to unpack its content during execution. The heuristic scanning method could report that this kind of behaviour is considered suspicious, as malware tends to do the exact same thing. Table 2.1 on the following page shows that 10 of 36 antivirus products believes that the patch is a malicious program, even if it is harmless and quite useful for users of the mIRC application.

What is in a name? Malware signatures are normally not standardized in naming across the different antivirus vendors, leading to a difficult process of aggregating and combine information from the sources releasing virus information [vir05]. The CARO virus naming convention was formed back in 1991 to address the naming problem [vir91], but even if the conventions now exist and have received extensions over the years [Sch99], they are not widely applied when new signatures are named. Consider the last example again; the legit patch program is detected as a threat on several antivirus products, but the signature name differs significant as the second column in Table 2.1 indicates.

Another problem with named virus signatures is that the information given is often spare, and many of them are of such general form that they hardly give any valuable or usable information at all. By again using the same example, signature names like “Generic Patcher”, “Trojan Horse” or “PAK_Generic.001” are generic names that have arisen from a heuristic scan. TrendMicro has the following description for the “PAK_Generic.001” signature from their open virus database³.

This (the signature name) is Trend Micro’s proactive detection for suspicious – and possibly malicious - executable files that are compressed using Win32 compression tools. This detection also encompasses many appending viruses found in the wild.

This heuristic detection is based on well-established characteristics inherent to packed malware. All Portable Executable (PE) files found on Windows 95, 98, ME, NT, 2000 and XP matching these estab-

²Located at <http://fish.sekure.us/BETA/fish130beta.zip>.

³Can be found at <http://www.trendmicro.com/vinfo/virusencyclo/default5.asp?VName=PAK%5FGENERIC%2E001%5C&VSect=P>.

Product	Virus signature name
AhnLab-V3	Win-Trojan/Agent.20480.KQ
AntiVir	TR/Agent.46592.C
BitDefender	Backdoor.Generic.58074
Ikarus	Backdoor.Pigeon.6620
K7AntiVirus	Trojan.Win32.Agent.Family
NOD32v2	Win32/Agent.OBH
Sophos	Generic.Patcher
Symantec	Trojan.Horse
TrendMicro	PAK_Generic.001
Webwasher-Gateway16	Trojan.Agent.46592.C

Table 2.1: Scan result showing false positives by 10 of 36 total antivirus products used. Only the positive results are shown in the table. The data is based on results gained from a file scan on <http://www.virustotal.com>.

lished characteristics are immediately detected. This keeps the customer one step ahead against possible virus infections.

Even if the above description is skim and does not dig into technical details, there are still statements to be aware of. The description underlines that a positive result will not guarantee a malicious file, but only state the possibility of it. So what will the user of the antivirus software believe? The description also says that files found are immediately detected, and the antivirus product will most likely report a positive result in the same way as more clear signatures do - even if the result is only a *possible* infection. Other than this, the description does not give information of any particular value due to its highly generic level, and the only useful detail about its functionality is that it is a compressed executable file. These reasons make it hard to base the analysis conclusion upon generic signatures compared to more directed signatures that are supplied with a concrete explanation of the functionality.

Two different styles of scanning Most antivirus products are featured with the possibility of starting a scan initiated by the user. This is called an *on demand*, or *manual* scan, and is the focus for this report. The other style of scanning is called *real time* or *on access* scanning, where user actions are monitored as they happen. Due to the large number of possible user actions, the real time scanning is subject to consume plenty of resources in the system. Since real time scans are most relevant to apply on threats to interactive events like browsing web content or opening email attachments, this scanning type will not be discussed further in this report.

2.2.2 Firewalls

The other set of proactive defence is an appliance which inspects network traffic and denies certain packets based on rules set by the user. The programs are

called *firewalls*, and can prevent malicious software from entering the machine. If the machine already got infected, the firewall can also prevent any attempt of malware “signaling home”⁴. There are several types of firewalls spanning from application level software to hardware based utilities. The firewalls can be based on packet attributes like source of origin and destination address, or states in the communication protocols [BC94]. Firewalls can certainly be used to prevent malware, but the required set of rules to stop all kind of malware is too large to manage a complete and updated set. Many state-based firewalls allows all established connections on the machine, but since it is often the user behind the keyboard that initiate actions leading to infections, firewalls become useless. If used correctly, firewalls can prevent attacks or infections, but the way they are normally used does not hinder the user to click on neither a malicious link nor visiting a web site filled with malware.

2.3 A brief introduction to NAAS

The NorCERT Artifact Analysis System (NAAS) is a web-based system used by the analysts from NorCERT as a central repository for storing information about malware samples. Only necessary details for the reader will be written in this report, as its design internals are to remain stricly confidential to avoid leaking valuable information.

NAAS is a collaborative tool and is used to share information across the analysis team. The software is partly implemented by in house software developers, and there is now a requested feature to support third party programs running through the regular NAAS interface. This feature is now being designed, and at completion there will be possible to hook a standalone program onto NAAS and run it as it is a component in the NAAS system.

⁴Malware tends to communicate with fixed servers (their “home”) with the purpose of receiving an updated set of instructions, commands or software.

Chapter 3

Malware analysis

This chapter covers a preliminary study of common methods and useful tools applied in the field of malware analysis. The analysis area is advanced, complex and large in scale, but this chapter will not enter deep technical details nor give a deep understanding of the field. The chapter will instead supply the reader with a required overview of the concept.

The chapter starts with Section 3.1 which gives an introduction to the analysis process, and continues with Section 3.2 with information about different methods used in an analysis. The chapter goes on with three similar sections, Section 3.3, 3.4 and 3.5 presenting applications commonly used by malware analysts during the different analysis phases. Table 3.5 on page 27 is made to give a quick overview of the described analysis tools.

3.1 The analysis process

When source code and documentation for a program is unavailable, understanding its complete functions and behavior may be difficult. Analysts are often set in this position, especially since authors of malware try to hide their code [KCV05]. There are two main approaches to solve this problem, called respectively *dynamic* and *static* program analysis. The first one studies the behavior of the program while active and running. Applying this method on a malicious program requires the analyst to closely observe program activity, including file system changes, network communication and suspicious collection of data and information from the system. If infected program files are executed during an analysis, the program might compromise the machine if the circumstances are correct, since this might be the goal of the malicious software itself. For this reason it is extremely important that the analysis process is under tight control and changes to the system state are reversed as soon as possible to sustain a secure environment. Malware today often try to communicate with host servers for downloading new malware or receive further instructions [DD07], so an Internet connection might be unwise as you not necessarily know what will go

through the communication network. On the other hand, in an analysis you *do* want to simulate a real infection so all effects are observed - including prior and after any suspect communication. There is no single correct answer of how to perform a malware analysis, but if the sample is allowed to run wild, the analyst has to ensure he or she can control it. To control the malware execution properly, the analyst needs as much information of the software as possible.

Dynamic analysis is usually applied before the static analysis, since the requirements in both resources and skills for static analysis are relatively high. Prior to dynamic analysis, some simple checks are usually applied on the suspect file to uniquely identify it. Hash sums, both MD5 and SHA are generated to ensure that the sample is not already known and analysed. If the sample is unknown, the analysis process continues. If the sample is already known and analysed, analysis process is complete and exits successful. During the dynamic analysis, the program is observed during its execution. Most analyses stops here, successful or not, due to the difficulty of the next process which is the static analysis. Some samples however require special treatment and are given the resources needed for this last analysis step.

While dynamic analysis has its similarities with proactive research, static analysis is more reactive where the goal is to deduce the functionality of the program by reversing the compilation process. This is called *reverse engineering* [Eil05], and applying static analysis methods like this one forces the analyst to work on byte code, which compared to high level programs like Java, Ruby and C++ is extremely low-level and complex. Producers of malicious code now write programs that obfuscates itself by using encryption and anti disassembling tricks and this will definitely not make the problem any easier [SF01].

Analysing malware is both time consuming and difficult. Compiled programs written by an attacker in a couple hours, may require days of analysing to reverse the process. A combination of both dynamic and static methods are therefore often required to understand the functionality of the program with a reasonable amount of resources available. As malicious software is getting more intelligent, the analyst is often required to think like an attacker and familiarise himself with obfuscation patterns to conduct a successful analysis.

The goal for this project is to add one additional step. The new step will in most cases appear just after the surface scan, but before the initialisation of the dynamic analysis. The content of the step is to scan a sample with an arbitrary amount of antivirus products that informs the analyst with valuable reports from all of the antivirus products. The new step will happen without any cumbersome manual intervention, as it will be automated as much as possible. Figure 3.1 on page 22 displays a graphical representation of the analysis process including the additional step called “antivirus coverage testing”.

This new step can also be integrated at a later stage, since new, possible harmful, files can be introduced at the dynamic analysis phase. These new files can be downloaded or unpacked by the malware, and in either way can contain hidden threats. Such files unpacked or downloaded are normally not camouflaged as good as the initial pack of malware, thus increases the probability of finding a matching signature. As scanning all possible new files during the anal-

ysis do not scale when done manually, it is a relatively simple procedure when applied as an automated process.

3.2 Analysis methods

Luckily, the analyst has some tricks up his sleeve which makes the fight against malicious software feasible. This section covers the most common and often used methods applied during the analysis.

Having total control over the analysis is essential to avoid further damage from the malicious software. If the machines used for analysis were to be infected and controlled by the attacker on the outside, there might be fatal consequences as the analysts reputation would be strongly crippled and valuable information could get stolen by an attacker. A dedicated room (a “laboratory”) might therefore be used during the analysis, physically sealing both analysts and the dangerous software from the rest of the world. Another more practical solution is to separate the networks with few entry and exit points, strictly controlling the network traffic to avoid any attacker from gaining access.

3.2.1 Separate networks

As malware tend to use the Internet for suspect activity, the laboratory has its separate network without access to the Internet. Files must be transferred by other means, and expected responses to malware trying to signal home must be simulated by the analyst in the best way possible.

3.2.2 Virtual environments and software based snapshots

When for whatever reason Internet access is needed, the analyst can prevent the malware infecting the host machine by running an operating system in a *virtual environment*. How this works will be discussed more in detail in Chapter 4, but needless to say this assists the analyst with tools to trap and deny unwanted behaviour by the malware. Additionally, it gives the possibility of reverting to a fixed clean state by software snapshots (system state image).

3.2.3 Hardware based snapshots

The virtual environments use software based methods for reverting a system state. This is however not the optimal case in all situations. By using hardware based snapshots instead, each system reboot loads the default state into memory. This will clear all changes applied on the machine, thus cleaning possible infections. Hardware based snapshot indicates a different approach to software based, by making the reversion process a default operation, instead of software based snapshots which is mostly applied on demand. Additionally, hardware based snapshots gives the analyst the opportunity of working in a non-virtual environment. Certain malware can detect if the system is running in a virtual environment and halt its own execution. This will be discussed in Chapter 4.

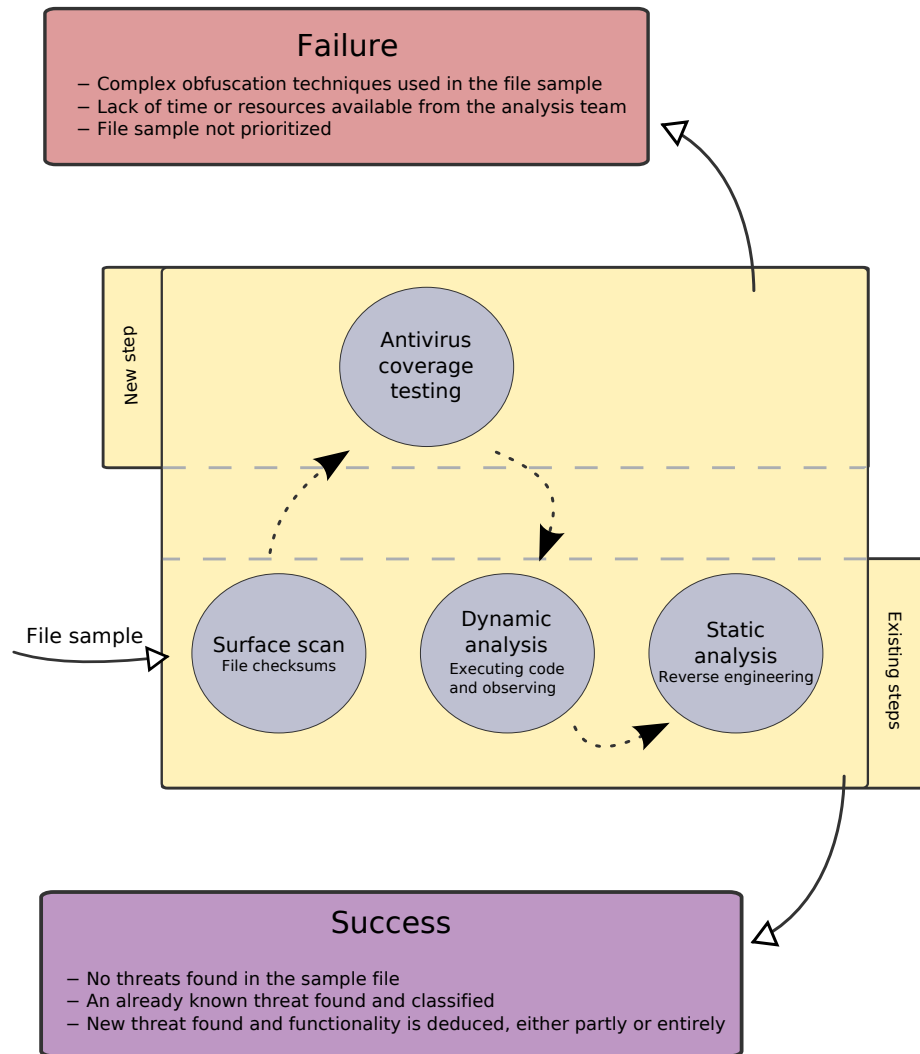


Figure 3.1: Graphical representation of the different steps of a malware analysis, including antivirus coverage testing which is the task for this project to arrange.

3.3 Surface analysis tools

When a malware sample is received, surface scans are most commonly the initial phase which gives the analyst a quick overview of the sample. The most common tools used are briefly described below.

Hash sums As a first identification mechanism, hash based values are generated from the samples. The result of a hash function is a fixed size string value which can with a large probability identify the uniqueness of an object. The most commonly used, and therefore publicly accepted, functions are MD5 and SHA. There exists theoretical attacks on the MD5 algorithm [WFLY04], but such attacks are not of any imminent threat to the surface scans. Even if it was, the SHA algorithm is also used to identify the sample which is stronger than MD5. The programs utilised are simply known as `md5sum`, `sha256sum` and `sha512sum` and generate a string length of respectively 128, 256 and 512 bits. A screenshot of these programs in action can be found in Appendix C.

PEid is a small, but important, application that detects most common packers used in the *Portable Executable* format [Pla07] (Mostly files with suffixes (.exe, .dll, .ocx, .sys and .scr). As malware often pack itself to avoid detection, this tool can be used as a first step of unpacking the software consisting the actual malicious functionality.

Office file scanners Vulnerabilities in the *Microsoft Office* programs often emerge, and since this office pack is so widespread, the vulnerabilities are valuable targets for malicious code. Normally, the exploits for the vulnerabilities require user interaction in form of opening a malicious file. Opening a file can be dangerous, but luckily there exist several tools that can be used to find suspicious abnormalities in the files. `MOICE`¹, `officecat` and `officebat` are all such programs. `MOICE` is kind of a wrapper which converts the binary code in the Office files to an XML representation. If unsuccessful, there is indications that the file is infected. `officecat` and `officebat` are command line tools used to scan files in a similar way, but unlike `MOICE`, these programs are not supported by Microsoft.

3.4 Dynamic analysis tools

The dynamic analysis phase usually initiates after the surface scan ends. This phase is contrasting the surface scan in the way that it consists of mostly manual work, with a handful of different tools available. They work as normal applications do, “on top of the operating system”. It is important to note that if malware has managed to breach deep in the operating system kernel and successfully altered the normal flow expected in an application, the tools used

¹Acronym for “Microsoft Office Isolated Conversion Environment”

might give erroneous information and contribute as a weakness for the dynamic analysis phase.

This section will only briefly cover the most relevant tools used in the dynamic analysis phase. Describing them all in detail could require the space of a thick book.

3.4.1 Sysinternals' system information utilities

Sysinternals' project consists of dozens of administration and diagnostic utilities for the Windows operating system. The company behind the software packages, which initially was founded by two software developers, was acquired by Microsoft in 2006 [sys06]. The software is freely available for download from their web page ². There are over sixty applications on the list of tools, and they include coverage for both troubleshooting and diagnostic tools, as well as monitoring the computer for internal activity and network traffic. When doing dynamic analysis, some of the tools can help keep track of changes to the file system or additions to the registry. In addition, malware often tries to open connections to foreign hosts, and the Sysinternals tools can give the analyst immediate notice. The most relevant tools for malware analysis are covered below.

TCPView

Imagine you are working on a computer which is suspected to be infected with malicious software. Malware is often working as a background process and is waiting for remote commands before the actual malicious events happen. Naturally, you are curious of which hosts the computer is sharing information with, or if it is listening for incoming requests. TCPView can be used to gain such information, and it is configurable to display the information you need, and strip away the unnecessary part. The TCPView application does however only list the connections, not the data traffic. In a way, TCPView can be described as a graphical user interface to structure and parse information from the program `netstat`, which is shipped with Windows installations. A screenshot of TCPView can be found in Appendix C.

Process monitor

To be able to spread through other processes, run at each system boot or store non volatile information, malware often use the file system and registry [Erd04]. To understand how a particular malware sample works, it is for this reason essential to know *what kind* of data being stored and *where* to locate it. After launching a binary file for dynamic analysis, it may or may not tamper with the computer - `Process monitor` assists the analyst with a list of changes done by the process. Since this is such a common procedure to expect from malicious

²Can be found at <http://technet.microsoft.com/en-us/sysinternals/default.aspx>

programs, `Process monitor` is one of the core utilities in the malware hunting toolkit.

3.4.2 Wireshark

Wireshark is a network protocol analysis tool which saw the bright of the day in 1998. The continuous improvement of this freely available and open source software, have made it strong, enriched with addons and a de factor standard for sniffing and reading network traffic. With the support for decryption of IPSec, SSL, WEP and Kerberos, even encrypted data can be read as plain-text with this powerful tool. The main usage of this application regarding malware analysis, is its basic data capturing properties. Compared to `TCPView`, it does not only show the active connections, but also the content of the data transfers. For this reason, **Wireshark** is a more in-depth tool for analysing network data. Due to its level of detail and lack of connecting network traffic to system processes, it is not used as a replacement for `TCPView` but rather as an addition. A screenshot of **Wireshark** can be found in Appendix C.

3.4.3 Sandboxes

The security mechanism called *sandboxing* is a way of safely executing programs in a strictly controlled environment. A malware analyst can utilise this power to deduce the functionality of a malware sample safely by having a jailed setup regarding network, disk usage and process spawning. This way of setting custom restrictions on the system might be required prior to running an unknown program. Several commercial sandbox-systems exists, but they are too complex to dig into for the scope of this section.

3.5 Static analysis tools

This section describes briefly various tools available for doing a static analysis process. The reason for giving less detail, is that the dynamic analysis is more relevant for this project than the static analysis process. The focus should be kept accordingly. Still, there is important for the reader to know that there exist tools also for the static analysis phase.

IDA Pro is a *disassembling* tool, and one of the key elements in each malware analyst's toolkit. Disassembling means transforming binary code into one of the lowest levels of code possible, merely the assembly machine code. **IDA Pro** makes the reversing process feasible, and the powerful debugger built-in makes it possible to analyse each single step of a program, breaking down obfuscation techniques. This application has been used to understand a large handful of complex malicious code by antivirus vendors, vulnerability research companies

and government agencies³. A screenshot of IDA Pro can be found in Appendix C.

Hex-Rays Decompiler approaches the problem in a different way. As IDA Pro is producing assembly code, this tool tries to transform the binary file to high level pseudocode similar to the C syntax. This will create code that in most circumstances is easier to read than assembly code.

BinDiff is used to compare two binary files, showing its differences. Malicious samples changes, and the surface scans (remember the first hash sum checks) gives complete different results indicating a never seen before sample. If the malware just changed slightly, this tool can help connecting a sample to an already known analysed sample.

strings lists textual information from data in a file. This is interesting when working with files containing *mostly* binary data. By using **strings** together with pattern searching, IP or email-addresses from a file can be found and useful information about the contents, and even functionality, of the file can be gained.

grep is a pattern matching program that is often used to search textual data for expressions or structured data. **grep** can be combined with any program that writes to a file or *stdout* (the command line display), makes it both powerful and easy to use.

³See <http://research.eeye.com/html/advisories/published/AL20010717.html> for an example.

Name	Comment
Hash sums	Various small programs used as an identification mechanism.
PEid	Used to detect runtime packed malware.
Office file scanners	Used to scan files in the Microsoft Office format for abnormalities.
(a) Surface analysis tools	
Name	Comment
TCPView	Used to monitor source and destination for network traffic.
Process monitor	Used to monitor file system and registry.
Wireshark	Used to analyse content of the network traffic.
Sandboxes	Complex systems for running dynamic analysis.
(b) Dynamic analysis tools	
Name	Comment
IDA Pro	Used to disassemble binary code.
Hex-Rays	Used to decompile binary code.
BinDiff	Used to compare binary files.
strings and grep	Small programs used to make common work more easy for the analyst during the static analysis phase.
(c) Static analysis tools	

Table 3.1: An overview over the analysis tools described.

Chapter 4

Towards a solution

With background in research described in the previous two chapters, this chapter explains in high level what kind of requirements there are for the system-to-be. The chapter covers existing possible solutions, and does also explain the virtualization technology. The chapter functions as a bridge and a dependent part to the next chapter, the requirements specifications.

The chapter starts with Section 4.1, which covers already existing possible solutions. The section discusses why the existing solutions are not sufficient, and continues with Section 4.2 which summarises requirements for the system-to-be in non-measurable high level terms. The last three sections discuss the virtualization phenomenon in a reasonably detailed level. The reader is encouraged to study these sections to understand how the technology the system-to-be will be based on works. However, the chapter is complex and there is no need to become too absorbed in the content, but instead use it as background material to understand the system-to-be.

4.1 Existing related work and possible solutions

There are several existing known solutions and methods for the problem description in Section 1.1 on page 5, but this chapter will conclude they all have certain negative sides, and a more suited alternative is needed.

4.1.1 Multiple antivirus products installed in the same operating system

As mentioned in Section 2.2.1 on page 13, many antivirus products support scanning active files on the computer and block access to the file if an infection is found. This kind of real time background scanning is feasible for the applications only when it can intercept system calls deep in the operating system kernel. This technique is called “API-hooking” [Hau07], and normally causes problems if more than one application is trying to lock the file.

Consider antivirus application “A” intercepting a “CreateFile” call to inspect for suspicion. The application will claim exclusive access to the file through the operating system. Antivirus application “B” also tried to intercept the call, but was not able to catch it before “A” did. “B” now notices that another program claimed the interception and could suspect it to be malware, as such API hooks are not common to see in legitimate programs. Still, the programs that try to ensure system security is the ones using the API hooks techniques. Clearly, there is a paradox.

Anyway, the claim for exclusive access initialised a fight between application “A” and “B”, and might end in a deadlock that might leave the system unusable [CES71]. This type of scanning can in most of the cases be disabled in the respective antivirus configuration, but most antivirus products notifies the user of existing conflicting antivirus applications prior to installation to be sure to avoid the problem. There are some tricks to avoid the installation checks and make it possible to install multiple antivirus products on the same operating system. However, the tricks are classified as rather nasty temporary hacks instead of a long term real solution. The antivirus installation programs are looking for conflicting products in some way. Due to the complex nature of an operating system, there are not one and only one way to do this. An obvious way of doing this could be to sequentially scan the file system to find any conflicting files. This approach could cause long delays in populated file systems, so the installers instead look for a combination of entries in the operating system registry and running services in the operating system to speed up the process. Since the process is probably considered “uninteresting design details” by the antivirus vendors, it is not likely to be found explained in their published documentation. After a manual try-and-fail process done for five products, the result was that all of them searched the entry `HKEY_CLASSES_ROOT\Installer\Products` for fixed hexadecimal hash values. When deleting or changing the entries for the particular product, some of the installations did not complain about conflicting products even if they were installed, and even active and running in the background. However, in addition to this check some of the installation processes also searched for running services and is a lot harder to get around without un-installing the entire product.

Table 4.1 on the next page shows conflicts discovered when trying to install the different products. The surd symbol ($\sqrt{}$) indicates a successful install, while a modified multiplication sign (\otimes) means the opposite. The left column shows the first installed product, and the next columns shows the product to be installed on top of this. Not all products were tested together (marked as “Not tested” in the table), but the table still clearly shows that there is hard to accomplish installing multiple antivirus products on the same operating system, and not feasible as a long term solution.

4.1.2 Virustotal

Spanish Hispasec Sistemas offers the use of a web based application called **Virustotal** that is able to scan submitted files on demand. With currently

Product name	McAfee VirusScan	Norton IS 2008	Panda Antivirus	Trend Micro IS 2008	F-Secure Client Security
McAfee VirusScan	-	⊗	✓	⊗	✓
Norton IS 2008	⊗	-	Not tested	⊗	✓
Panda Antivirus	Not tested	⊗	-	⊗	Not tested
Trend Micro IS 2008	⊗	⊗	⊗	-	⊗
F-Secure Client Security	⊗	⊗	Not tested	⊗	-

Table 4.1: Conflicts between antivirus products on the same operating system.

36 available antivirus applications, **Virustotal** seems to be the de facto standard in the security business for on demand scanning with the use of a large number of antivirus products. Analysts from NorCERT often use this service, but **Virustotal** cannot be used at all times. To increase each antivirus product detection rate on malicious files, **Virustotal** shares submitted files with antivirus vendors¹. If the submitted file is sensitive malware targeted against a particular firm or company, there is not necessarily a good idea to spread the software further to these non-trusted third party firms. Software samples might even be classified, making uploading to the Internet illegal by law. Since the **Virustotal** application is closed source and owned by a third party company, any requested changes to the software might be hard or impossible to accomplish. Finally, as **Virustotal** is web based, an Internet connection is required to communicate with the service. Internet access is not necessarily available when analysing files in an isolated and closed network. As there exist numerous frameworks with web interactions, automatic interaction with **Virustotal** should however be quite easy to accomplish, but due to **Virustotal**'s other characteristics, use of this service cannot be accomplished in all circumstances and is eliminated as a possible solution.

4.1.3 Metascan

Metascan is currently a Windows-only application supporting on demand scanning of files by six built-in antivirus products. In addition to this, the program can use antivirus products installed in the same operating system. **Metascan** is therefore a local version of a small **Virustotal**. The application requires interaction through a GUI, so automatic interaction could therefore be non-trivial to accomplish with the program. The amount of built-in antivirus products is

¹See **Virustotal**'s privacy policy at <http://www.virustotal.com/privacy.html>.

also too small to accept. **Metascan** allows extending the number by using the already installed antivirus products on the system. As seen in Section 4.1.1 on page 29, this might not necessarily be a trivial task either. Extending the number of products from six to seven seems pretty straightforward as you only have to install one antivirus product on the machine². Extending from seven to eight antivirus products introduces a lot more problems. Due to **Metascan**'s lack of scalability, it is eliminated as a usable solution.

4.1.4 PowerScan

The dynamic analysis framework, **PowerScan**, was designed and developed during spring 2008. Similar to this project, **PowerScan** was also in cooperation with NorCERT. The results are based on a problem description similar to the problem description in this project, but there are several important differences. **PowerScan** covers a larger and more general spectrum of dynamic malware analysis, and the implemented solution is meant as a framework rather than a finished tool. According to the published master thesis [LL08], the developed software was written in Java, but it was not available at the time of writing this report. The possible benefits of using the **PowerScan** software would be minimal for this project due to a different project scope. Some of the **PowerScan** research are still useful for this project. All places the work from the **PowerScan** thesis is used, are clearly marked to credit the authors and avoiding any plagiarism.

4.2 High level requirements for a new solution

While none of the known available solutions are sufficient for a new solution, their positive properties can be combined to a new and better solution. Some high level requirements for the system-to-be are therefore listed below to give an indication of how the system design eventually will be. Each requirement is labeled for a referring purpose. More concrete requirements are elicited in Chapter 5.

HR.01 The system should be automated in such a way that an analyst is able to submit a file to the system and receive the results without any more required effort.

HR.02 The returned result from the system should be presented in a way that makes it simple for a person to read and easy for a machine to parse. The last property is relevant if the system is to be integrated with another system.

HR.03 The system should be scalable in such a way that additional antivirus products may be added as needed with no more than a linear increase in running time.

²The product has to be detected by the **Metascan** software, but most of the common antivirus products are supported.

HR.04 Software updates for the antivirus products should be supported, preferably in an automatic way.

HR.05 The system should support remote communication so the hardware intensive tasks can be distributed.

HR.06 The expenses for hardware and software is not an issue and not prioritised. However, the running time should be kept as low as possible, but no time limits are set.

HR.07 The software should be released as open source and be freely available. This requirement will make the project unique compared to existing solutions, as more people will have the possibility to freely use it and contribute new features as needed.

HR.08 The released software should be documented so possible contributors more easily can familiarise with the code.

The above list utilises the submission based form and remote communication *Virustotal* offers, the simple addition of new products as *Metascan* partly offers (remember the conflicts from different concurrently installed products) and parseable data and automatic updates as seen in theory from the *PowerScan* work. The open source requirement will be a unique requirement for this project, as there are no known existing similar products open for contribution or public download³.

4.3 Platform virtualization

By using several operating system installations, you can get around the problem regarding antivirus products causing conflicts to each other. Normally, each active operating system requires unique access to hardware such as CPU and memory, but utilising a technology called *platform virtualization*⁴ gives the possibility of sharing hardware devices across different operating system installations. In this way, the operating system is “fooled” to believe it has unique access to the computer’s resources, even though there might be an arbitrary number of operating systems installed and actively using the computer’s resources simultaneously.

The virtual machines (VMs) act and look like a real computer, but as Figure 4.1 on the next page shows, they are running inside another operating system. If each *virtual operating system* has its own antivirus client installed, there will be no interference with products installed on other virtual systems. This requires the underlying virtual machine system to deny the interference across the

³Analysts from NorCERT are not familiar with the existence of any such project, nor did any searches through the open source contributing channel SourceForge give any relevant results.

⁴Several other types of virtualization exists, but none of them are relevant for this project. *Virtualization* will therefore henceforth be used to signify platform virtualization.

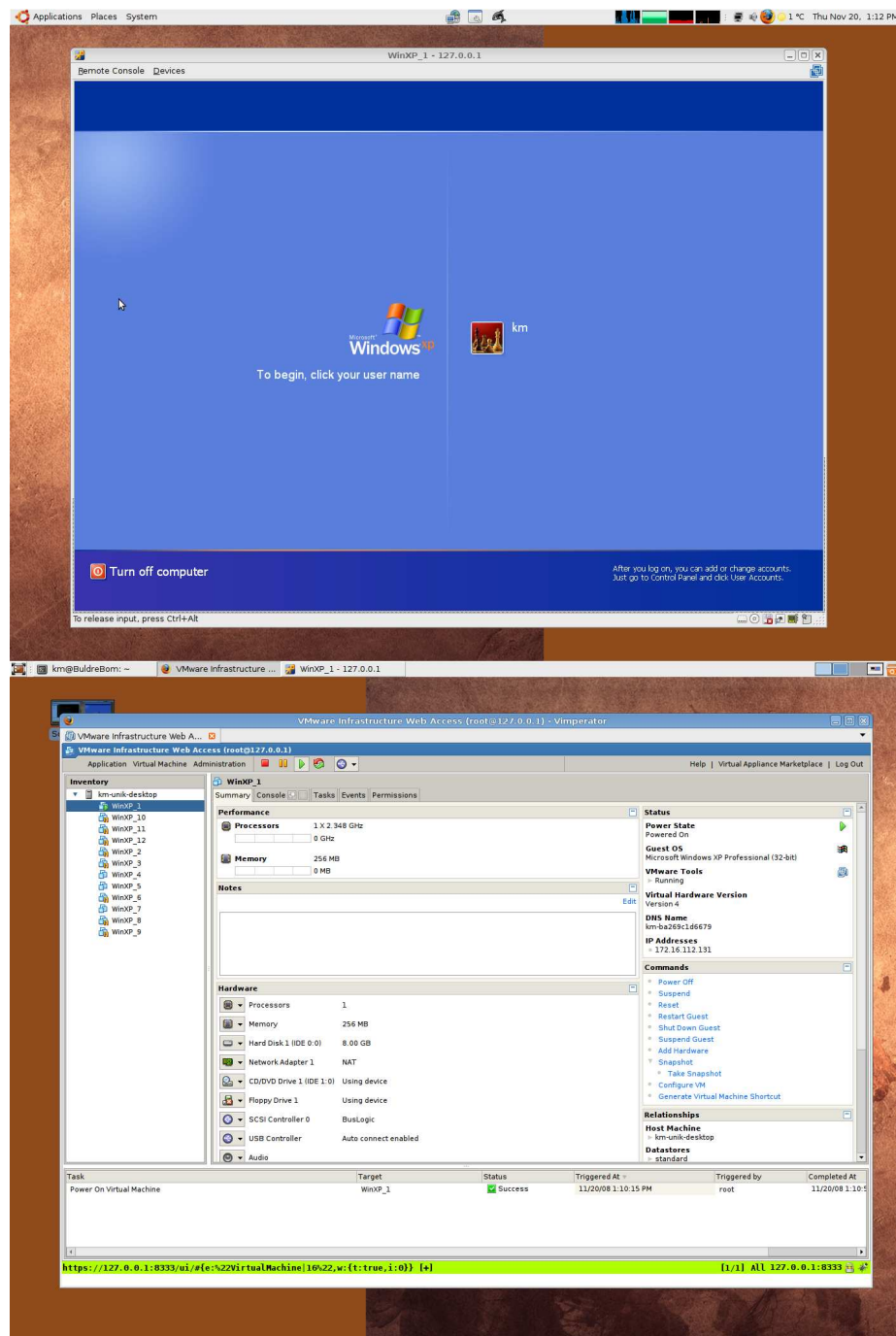


Figure 4.1: Screenshot of a Microsoft Windows XP operating system running from an Ubuntu operating system installation. The upper part of the screenshot shows Windows XP (its login screen) as a “windowed application” running in Ubuntu. The lower part of the screenshot shows a control panel to administrate the virtual machines. The screenshot is not meant to give any details, but merely an overview to show the concept of virtualization.

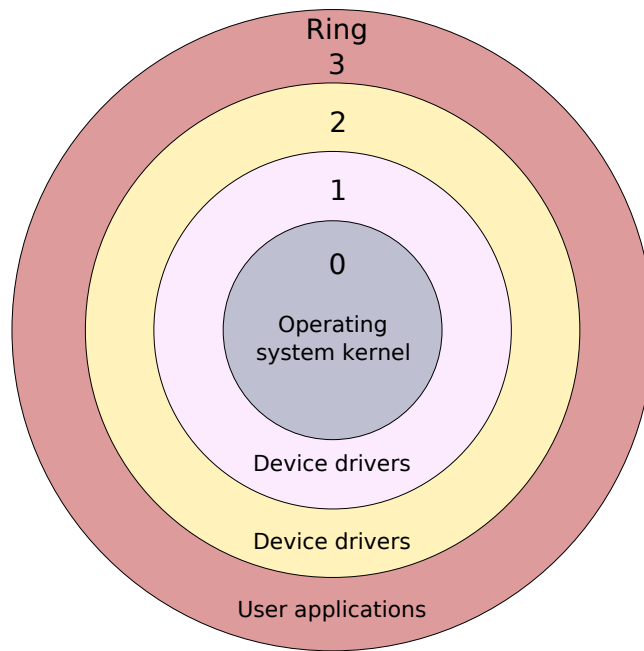


Figure 4.2: Graphical display of the different protection rings. The figure is based on Intels IA32 protection rings [ext01].

different virtual machines. To achieve this, a “hypervisor” (also called a virtual machine manager) is running in the same *protection domain* as the operating system kernel and distributes hardware resources to the virtual machines.

4.3.1 A brief explanation of hierarchical protection domains

As first pointed out for the Multics operating system, a mechanism to group system parts into different protection domains called “rings” can enhance the security of the system [SS72]. It is structured hierarchical, where “Ring 0” is the most privileged and trusted level, “Ring 1” has lesser trust and so on. The operating system expects to be able to run instructions in the most privileged ring, and to have direct access to the hardware. The processor architectures from leading producers AMD and Intel are both applying similar privilege grouping of instructions, and virtualization techniques must therefore be designed to accomplish cross border translations of instructions in the rings. Figure 4.2 displays the hierarchy in a graphical way where the operating system kernel is running as the inner ring, user applications in the outer ring and device drivers between.

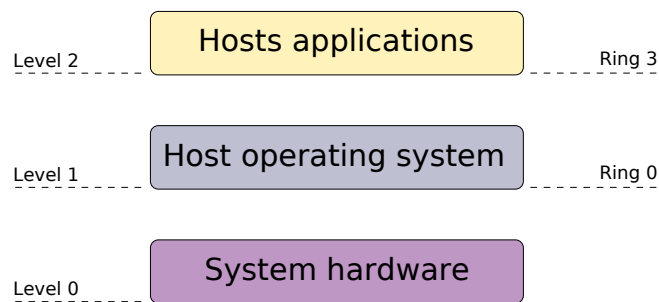


Figure 4.3: Architecture diagram of a computer with one operating system.

4.3.2 Different styles of (platform) virtualization

There are currently four main ways of applying virtualization, each with positive and negative sides. Three solutions are based on pure software power, while the fourth is utilising hardware to reduce necessary work by software [vmw07].

Full virtualization

Full virtualization is a software layer which aims to offer a complete *simulation* of the underlying hardware. This allows software to execute normally above the virtualization layer, and operating systems can run unmodified as having its own physical machine. Figure 4.3 shows a high level computer architecture as normally expected to be, and Figure 4.4 on the facing page displays the new elements to the system architecture. The difference between system levels and rings are how respectively the user and the operating system see the software access levels.

The hypervisor is included in the same ring as the normal user applications, but a *driver* is installed in ring 0 together with the operating system kernel [ext01]. The hypervisor can then run high privileged instructions and map system calls from the “guest” operating system (the guest operating system(s) are installed in virtual machines, communicate with the host operating system) directly on the hardware by communicating with the driver. A new level is introduced on top of the user applications which run in ring 3 and contains an arbitrary number of guest operating systems. All of those are taken proper care of by the hypervisor which job is, among other tasks, to ensure that interference across the virtual machines are prevented [PG74].

Partial virtualization

As the name indicates, *partial* virtualization only provides gradual simulation of the underlying hardware. This kind of virtualization do not normally allow a complete virtual operating system to run, but instead smaller software programs.

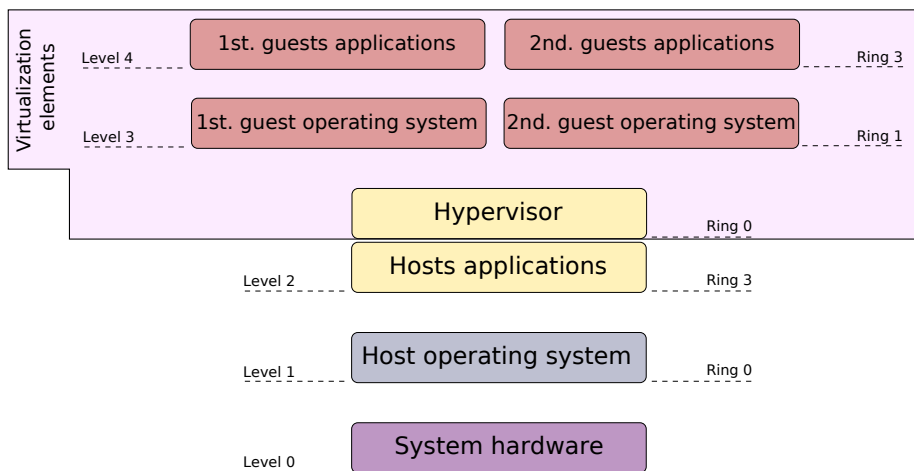


Figure 4.4: Architecture diagram of a computer with one main operating system with two virtual guest operating systems installed on top.

Paravirtualization

Paravirtualization introduces a software interface that is more similar to the raw hardware than the hypervisor software. This is done to increase performance by reduce the amount of interrupts sent to the host operating system and instead direct them to the underlying hardware. Paravirtualization allows guest operating systems to run in the most privileged ring. There is no way to manage this without modifying the operating system kernel, and currently there does not exist any such modifications to the Windows kernel. Several Unix based systems do, but not everybody might allow such tampering with the deep software architecture of their servers.

Hardware assisted virtualization

The original x86 processor architecture was not designed to meet virtualization standards, but in 2005 the competing chipset producers Intel and AMD made changes to their architectures to support virtualization-friendly processor instructions by adding a new privilege level to the architecture *below* ring 0 [Rub06]. By letting the hypervisor software run in this new ring (“Ring -1”), the guest operating system allows to run in the most privileged ring, and the high performance losses from the hypervisor are reduced. However, a study done by Keith Adams and Ole Agesen shows that there currently are little practical improvement of using the hardware assisted virtualization over classical pure software virtualization [AA06].

4.3.3 Why virtualize?

Placing different operating systems in virtual environments give the advantage of utilising idle hardware resources further than in a normal system. Both money and energy costs will be reduced if the number of servers is reduced and services are instead placed on the same machine [wra]. By introducing the *snapshot* functionality, which is a mechanism to store a system state, virtual environments can act as an simple rollback function for a complete operating system. The system created during this project will handle large numbers of malicious samples, and it is essential to sustain a secure environment. Since damage done by malware can be impractical reversible in a normal system, the snapshot functionality is a strong reason to use virtualization. After the completion of a job including possible malicious files, the clean and safe state can easily be restored from snapshot files. Another important property of using virtualization is the possibility of automated job processing, which is one of the high level requirements stated in Section 4.2 on page 32. Normal systems do not have this functionality, but some virtualization systems have this in their core. This will be further discussed when introducing the *VIX-API* in Section 4.6 on page 41. These two properties are powerful alone, and even more powerful combined. Job automation is the main reason virtualization outranks any other alternative.

4.3.4 Virtualization’s negative sides

Even though virtualization seems to be the way to go, it introduces some problems which need to be handled the best way possible.

Malware halting in virtual environments

Due to the unique opportunities virtualization give, analysts are often working in virtualized environments when analysing malware.

Even if virtual environments are meant to act like a complete replica of a normal computer, research has shown that there are some tricks available to detect the environment [GAWF07, QS00]. Malware programmers use the tricks to avoid the normal malicious execution if a virtual environment is detected, and instead halt the malicious program. This will stealth the real functionality of the malware, and avoid exposing its functionality to the analysts working in protected virtual environments. If the analyst even notices the irregularity, the malware forces him to move the dynamic analysis to a normal operating system and the reversing process with software based snapshots is no longer an option.

The few lines of code in Listing 4.1 on the next page is an implementation written in C by the Polish security researcher, Joanna Rutkowska. These lines are one of the results from her “Red Pill project” [Rut04], and is one code example which is able to detect if the operating system is running in a virtual machine or not⁵.

⁵There are some limitations, but the example is just meant to show what kind of methods

Halting of a program requires the actual program to run. Dynamic analysis may fail if a sample has a Red Pill implementation or similar in its core. Static analysis and on demand virus scanning on the other hand do not run the programs and halting of code is therefore not relevant for these phases.

```

1 int swallow_redpill () {
2   unsigned char m[2+4], r pill [] = "\x0f\x01\x0d\x00\x00\x00\x00
   \xc3 ";
3   *((unsigned*)&r pill [3]) = (unsigned)m;
4   ((void(*)())&r pill)();
5   return (m[5]>0xd0) ? 1 : 0;
6 }

```

Listing 4.1: C code returning a non-zero value if a virtualized environment is detected. Only a few lines of carefully selected code is necessary for a detection.

Increasing the amount of licenses

The hypervisor software is often proprietary and expensive. Handling these licenses comes with a price in money. There exist open source freely available alternatives, but these might not have the required functionality. Read more of this in Section 4.4.

Complexity-rise of each machine

Even if virtualization may reduce the amount of machines significantly, each virtual machine will still introduce a new layer of complexity as seen in Figure 4.4 on page 37. The new level of complexity may include increases in managerial costs and in time learning the software. On the other hand, the reduction in administration costs by reducing the amount of servers may weight this up.

4.4 Virtualization software

There is a solid handful of existing virtualization software⁶, and this section will cover the requirements for the virtualization software to use in this project.

At first it may seem difficult to select the best suited virtualization software for this project, but there are some important considerations to have in mind. First and most important, it must support job automation and remote calls. Secondly, the virtualization software should be stable and properly tested. Not all products supports these requirements, but the hypervisor software from *VMware* does, and will be selected due to this fact. VMware are dominating the virtualization market⁷, and widely used. A large online forum based user

there are available for detecting virtual environments.

⁶See <http://virt.kernelnewbies.org/TechComparison> for a list of examples.

⁷See as an example the two articles at http://searchservvirtualization.techtarget.com/news/article/0,289142,sid94_gci1225412,00.html and http://www.infoworld.com/article/07/08/14/VMware-the-bright-spot-on-a-gray-Wall-Street-day_1.html.

community is also available to help if problems arise during the implementation. Virtualization software from VMware supports handy techniques as cloning of operating systems, job automation and clever administration interfaces. A Windows operating system is often required by the antivirus products, but the software supports all relevant Windows operating systems. On the negative side, the software from VMware is proprietary licenced and controlled exclusively by VMware so any changes to the code is probably impossible to do.

4.5 Differences in the VMware software packages

The software assortment available from VMware is huge, and their webpages list around 15 complex products but only three of them are interesting for this use⁸, as the rest are meant more for administrative system infrastructures. The relevant products for this project are shown below.

- VMware Workstation
- VMware Server
- VMware ESXi

VMware ESXi is built with its own operating system and thus does not require a pre installed host operating system to run, but the job automation API which is discussed in Section 4.6 on the facing page does not support this product. Therefore, the VMware ESXi product need no further consideration as the time required to implement a new automation API is definitely not available in this project.

The workstation as well as the server version of the VMware products support the fundamental functions needed for this project, but they have some slight differences as we will see. The independently driven information site <http://virtualization.info> has published a feature comparison about the two products⁹. The most relevant points from this document are extracted and listed in Table 4.2 on the next page. Each of these points are worth a brief explanation. Initial launch of virtual machines requires a substantial time compared to other operations, so an automatic startup is a positive property. However, since reboots will not happen frequently, this is only a minor detail. Both products support the programming API, making them the two exclusive products for our selection. The snapshot functionality is valuable in this project, as the virtual machines can be reverted back to a clean state if it is suspected to be infected. The VMware Server edition does not support multiple concurrently stored snapshots, but there are no indication that more than one snapshot per virtual machine will be required. Only one “clean state” snapshot is needed for each virtual machine. A more important feature from the Workstation-version is

⁸Full product listing and further information can be found at http://www.vmware.com/products/product_index.html.

⁹Available at http://www.virtualization.info/lab/VMwareWKS60_vs_VMwareSVR10.pdf.

	VMware Workstation	VMware Server
Launch virtual machines at system boot	⊗	✓
Programming API	✓	✓
Programming API calls	Local only	Local and remote
Multiple snapshots	✓	⊗
Virtual machine cloning	✓	⊗
Pricing	Around \$200	Free

Table 4.2: Important differences in the two VMware virtualization products of current interest.

the support for cloning virtual machines. If a large number of antivirus products are planned to be integrated in the system-to-be, an equal amount of virtual machines need to be created (each having its own operating system and antivirus product). One Windows XP installation requires around 30-40 minutes, while cloning a virtual machine and its installed operating system takes around one minute. This could be a significant time saver.

Only the server version supports connections from remote machines, and having all available virtual machines locally will not necessarily be the case. The server version will therefore be selected as the software suite as an overall handler (installation and graphical manager) for the virtual machines. **VMware Server** is free software (free of cost, sadly not free as in open source “freedom”), and getting around the lack of cloning capabilities is possible. See Appendix A for more information.

4.6 The VMware Programming API (VIX)

VMware delivers both **VMware Workstation** and **VMware Server** installations with a programming API called *VIX*. The API supports high level programming for automation of different common tasks in the languages C, Perl as well as COM based languages (Microsoft Visual Basic, C# and VBScript). Two open source projects extends the list by providing bindings for Java (*jvix*) and Python (*pyvix*) [wha08].

Prior to select *VIX* as an automation framework, the high level requirements discussed in Section 4.2 on page 32 must be considered as basic functionality and *VIX* must support this. The *Programming API Programming Guide* [vmw06], which is available online, indicates that *VIX* got wrapper functions for all needed functionality applied on the virtual machine’s antivirus installations. Table 4.3 on the next page links the functionality from the high level requirements to the respective *VIX* function calls. Notice that in this design the running antivirus program has to write the results to a file so they can be copied back to the host operating system for parsing. Additionally, the antivirus program must support command line scanning (no graphical interaction). The last limitation will be discussed further in Section 6.6 on page 67.

Functionality	VIX function call
Initialise communication with virtual machine.	<code>VixHost_CONNECT()</code>
Start the virtual machine (Power it on).	<code>VixVM_PowerOn()</code>
Log on the virtual machine's operating system.	<code>VixVM_LoginInGuest()</code>
Transfer sample file to be scanned to the virtual machine.	<code>VixVM_CopyFileFromHostToGuest()</code>
Run an arbitrary program on the virtual machine.	<code>VixVM_RunProgramInGuest()</code>
Transfer the results back to the host machine for parsing and aggregating.	<code>VixVM_CopyFileFromGuestToHost()</code>
Revert the virtual machine into a clean state.	<code>VixVM_RevertToSnapshot()</code>

Table 4.3: Functionality needed in the design and the respective VIX function calls available to use.

Chapter 5

Requirements specification

This chapter focuses on elicitation, analysis and specification of the requirements for the implementation part of this project. A top-down approach to the requirements analysis is presented. This approach refers to a development starting basically from scratch, but utilise the existing high level VIX-framework as a building block. Within the requirements specification phase it is quite important to describe the application's qualities that has to be assured, rather than describing how such qualities will be designed or implemented ("what" versus "how"). Functionalities that the system has to supply are defined in this part of the report. This is done without indicating a specific architecture or a particular algorithm to adopt in the implemented solution. Software requirements analysis is one of the key elements of the development work flow as the other stages are based upon it, such as software design. The chapter flows by capturing functional requirements starting in Section 5.1. Next, the chapter describes several realistic situations in Section 5.2 which will eventually lead to a collection of formal requirements for the system-to-be, shown in Section 5.3. The chapter ends in Section 5.4 which maps the requirements to the high level requirements.

5.1 Use case analysis

The purpose of this section is to elicit important *use cases* [Fow03]. Use cases are used to capture functional requirements of the system to be implemented. The use cases are described in a textual form underlining the involved actors, the goal for the use case, the priority, pre-conditions, post-conditions, and basic flow of events. The latter point includes possible alternative paths and exceptions that may raise during the normal course of events. The goal describes what the actor wants to achieve with the current use case. Preconditions and postconditions are used to imply respectively requirements for the use case to happen, and events after the use case completes. Use cases are specified only if the condition is not considered trivial. Use cases assists in at least three areas, each listed below.

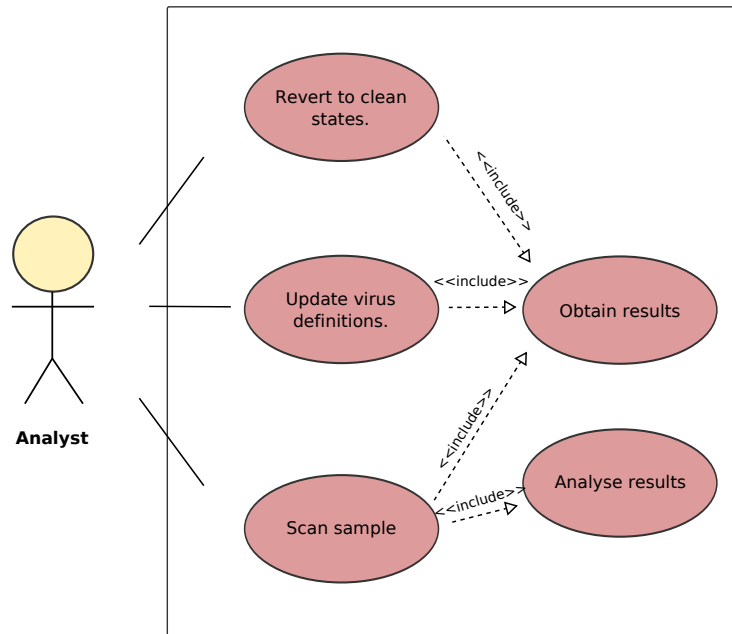


Figure 5.1: Overall use case diagram.

- Use cases can assist to process of deducing the starting set of requirements.
- Creating and analysing use cases can help determining overseen requirements.
- The simple notation assists the communication with supervisors and contact persons at NorCERT.

Most of the following use cases are supplied with a use case diagram to graphically display the textual representation. Together, they are considered as the starting point for the requirements specification. Investigating the use cases and their relationships makes it possible to more easily understand which are the most important services and functionalities that the system shall¹ supply. Implementing the described operations, moreover, is the best way to fulfill the expressed requirements.

An overall use case diagram is shown in Figure 5.1. This general use case will be split into smaller pieces in the use cases that follow.

¹RFC2119 [Bra97] is used in this chapter to indicate requirement levels.

5.1.1 First use case, “Scan sample”

The analyst needs scan results from all available antivirus products early in the malware analysis phase to avoid spending time on analysing samples already known by antivirus products. The analyst can also use the scan results to give an early indication of how dangerous the software actually is. The use case diagram for this case is shown in Figure 5.2, and explained further in Table 5.1.

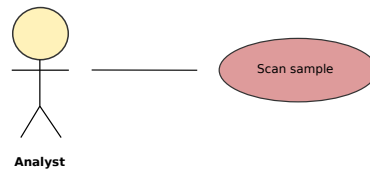


Figure 5.2: Use case diagram for “Scan sample”.

Name	Scan sample
Actor	Analyst
Goal	<p>The actor intends to use multiple antivirus products when scanning a sample and use the results:</p> <ol style="list-style-type: none"> 1. As an indication of what kind of malicious code it contain, if any. 2. Monitor the detection rates by the different antivirus products.
Priority of implementation	High
Entry conditions	<p>Virtual machines running and with operating systems and antivirus products installed.</p> <p>A sample file is available.</p> <p>A command line shell is available (Through terminal, SSH or similar).</p>
Flow of events	<ol style="list-style-type: none"> 1. The actor starts the command line script with a sample file location as argument. 2. The system presents information and scan progress on the fly. 3. The system presents aggregated scan results.
Alternative flow	<ol style="list-style-type: none"> 1a. The actor adds a “progress”-argument to the command line script. The system will present results not only at the end of execution, but during runtime. 2a. The actor cancels the operation and the use case ends.

Table 5.1: Use case for “Scan sample”.

5.1.2 Second use case, “Submit sample through NAAS”

While similar to the previous use case, this one differs by introducing a user interface through the NAAS system. The same diagram as the last use case had applies here, but includes updated information shown in Table 5.2.

Name	Submit sample through NAAS
Actor	Analyst
Goal	Similar to the use case “Scan sample” shown in Table 5.1.
Priority of implementation	High
Entry conditions	All conditions from Table 5.1 apply, and in addition; The NAAS integration module is loaded and active in the NAAS production environment. The NAAS system is available for the actor (he or she is authenticated, authorised and logged in to the system).
Flow of events	<ol style="list-style-type: none">1. The analyst selects the path for a (suspicious) file in the upload form and clicks the submit button for upload.2. The system presents a loading indication while the antivirus products are working in the background.3. The system presents aggregated scan results in the NAAS interface.
Alternative flow	

Table 5.2: Use case for “Submit sample through NAAS”.

5.1.3 Third use case, “Revert to clean states”

The analyst needs to revert virtual machine states back to a clean revision. The reasons for this are one or more of items in the list below. Figure 5.3 shows the use case graphically, while Table 5.3 explains the use case more in detail.

- There is reason to believe that the system has been compromised by malicious code.
- One or more of the virtual machines have in some way crashed or do not respond to operation requests. Manual intervention would require a graphical connection to the host and debugging the problem could be too expensive of current interest.
- Some change was made to the virtual machine state that is expensive, hard or impossible to reverse by using the same kind of methods.

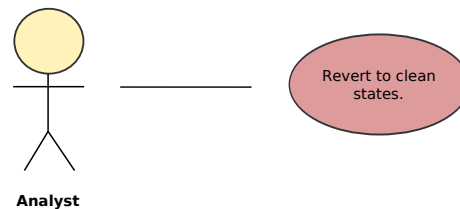


Figure 5.3: Use case diagram for “Revert to clean states”.

Name	Revert to clean states
Actor	Analyst
Goal	The systems need to be restored to a clean state.
Priority of implementation	Medium
Entry conditions	Virtual machine images are installed. There is reason to believe that the operating system is compromised or the environment on the virtual machine is disordered or non-working.
Flow of events	1. The analyst is starting the revert process by supplying the system program with a pre defined argument. 2. The system presents progress on the fly.
Alternative flow	

Table 5.3: Use case for “Revert to clean states”.

5.1.4 Forth use case, “Update virus definitions”

The virus definitions for each antivirus product receive subsequent regular updates, some several times each day. This is the way of feeding the products with new signatures which is able to catch newly introduced malware. It is therefore essential to keep the antivirus products up to date, and if possible this shall be performed in an automatic way. Figure 5.4 shows the use case graphically, while Table 5.4 further explains the use case.

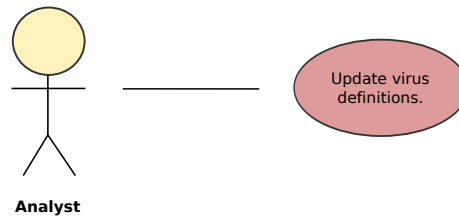


Figure 5.4: Use case diagram for “Update virus definitions”.

Name	Update virus definitions
Actor	Analyst
Goal	Each vendor releases new antivirus signatures quite often, maybe several times a week and all virtual machines should be automatically updated.
Priority of implementation	High
Entry conditions	Virtual machine images are installed. An antivirus product is installed on the virtual machine.
Flow of events	1. The analyst is starting the update process by supplying the system program with a pre defined argument. 2. The system presents progress on the fly.
Exceptions	1a. The update process fails. 1b. The system presents the results. 1c. The user case ends.
Alternative flow	

Table 5.4: Use case for “Update virus definitions”.

5.2 Some use case scenarios

A use case scenario describes a relevant real-world example of how people or organisations interact with a system. The scenarios describe the steps, events, and/or actions which occur during the interaction. The following scenarios are reasonably high-level and indicate how someone works with the system, but intentionally exclude technical details. The detail level depends on the amount of information needed to well understand the specified requirements. The figures connected to the scenarios follow the UML sequence diagram notation [Fow03], with the purpose to display a dynamic vision and interaction flow through the system. The diagrams show the scenarios that may exist at run-time when objects interact to achieve some particular goals. The diagrams display the progression of the time along the vertical axis, highlighting the temporal sequence of the exchanged messages among objects. The reason for using such a notation is to favor a more pleasant overview of the behavior for the reader. In addition, by analysing such diagrams, he or she can confirm if the requirement specification successfully capture the expected behaves. As the scenarios are all connected to same same system, the sequence diagram shown in Figure 5.5 on page 52 has been created to show the three first scenarios together. A separate diagram is drawn for the last scenario, shown in Figure 5.6 on page 53. The use case scenarios have a one-to-one mapping to the use cases described in Section 5.1. Each scenario have been thought within a story context and here below narrated.

5.2.1 First scenario

The scenario actor is Olvan Wangleflesh.

Olvan is one of the key players for the respected national group of malware prevention located in a foreign country. One of his tasks includes analysis of incoming malware samples, and nowadays the amount of suspicious files is tremendous in size. Analysing the samples consume a large percentage of Olvan's available time, too much in his opinion. The first steps of the analysis process are more or less the same with each sample, and this tedious process could really benefit from some kind of automation. Olvan already has his surface scans² automated, but this is not enough. "If I only could be relieved from checking the samples with the antivirus products, I would be satisfied for now" he says, with a small groan reminding him how low the antivirus detection rates often are - requiring him to often apply a large multiple of products on each file. Luckily, a new automated virus scanning system is now available. It is small in size and free for download. He downloads the package and starts reading the documentation. Olvan is already a bit familiar with virtual machine technology, which is a positive ability when using this new software. He locates an usable machine with a good chunk of memory from his closet, and in an hour he manages to set up

²Read more about surface scanning in Chapter 3.

seven virtual machines - each with its own antivirus installation. He is ready for testing the system and starts the program. He choose the file scan option, and supplements a sample file to be scanned. As the progress is shown on the screen, he observes that only two of the seven antivirus products found the file suspicious. He notes down the poor detection ratio and continues the analysis. The thought of all hours that will be saved from now on really pleases Olvan.

5.2.2 Second scenario

The scenario actor is Snook Yjcorpus.

Snook is a coworker of Olvan and immediately gets the good news. He wants the newly discovered software to be part of his NAAS web system, which is used to store information about malicious code and its respective functionality. He is glad to hear that the software outputs a way his NAAS system can read, and as skilled in software development, he quickly writes a short module utilising the power of Olvan's work and thus integrating Olvan's new solution into the existing NAAS system. By using the NAAS system as a portal for the automation system, the analysts can utilise graphical interfaces instead of command line switches. Snook locates a legit sample file and clicks the submit button in the NAAS system. After a short while, the system reports that none of the antivirus products found the file suspicious, indicating a harmless file - or just a really bad antivirus detection ratio.

5.2.3 Third scenario

The scenario actor is Olvan Wangleflesh.

Olvan suddenly realises the possibility of having his virtual machines infected with one of the malicious samples frequently uploaded to the system. Olvan is aware that the uploaded files are not being executed on the virtual machines, but still, the possibility for an infestation is unavoidable. He plans to disable the newly implemented system, but he then discovers another feature in the program - a kind of rollback functionality. "I knew I should properly read the documentation", he groaned. He initialise the program and supplements an argument to create snapshots for each virtual machine as the first step of trying out the newly discovered functionality. Olvan changed the virtual machine state by switching the standard desktop background from the blue default one in Windows XP to a more snow filled landscape picture (to better suit the current weather). After this visual state change, he runs the software with a new argument that is said to revert back to the time last snapshots were taken. To his amusement, he observes that the background is changed back again - the rollback functionality seems to work like a charm and can be applied whenever the virtual machines are subject to infestation.

5.2.4 Fourth scenario

The scenario actor is Olvan Wangleflesh.

Olvan has just finished reading the entire documentation for the new system, and notices the support for on demand updates for the antivirus products. Just as the capability of reverting to clean states, Olvan had not thought of this important requirement. In the machine filled with 8GB of memory, all the operating systems in the virtual machines are active and running simultaneously and thus automatically receive updates. At the time the amount of virtual machines are further extended, the memory in the host machine might be filled to the limit. To make room for the rest of the virtual machines, memory has to be released by temporarily shutting down the finished virtual machines. Olvan wants to try the updating process, and to his joy he only needs to supply a command line argument to the program. As progress shows, all the products receive updates.

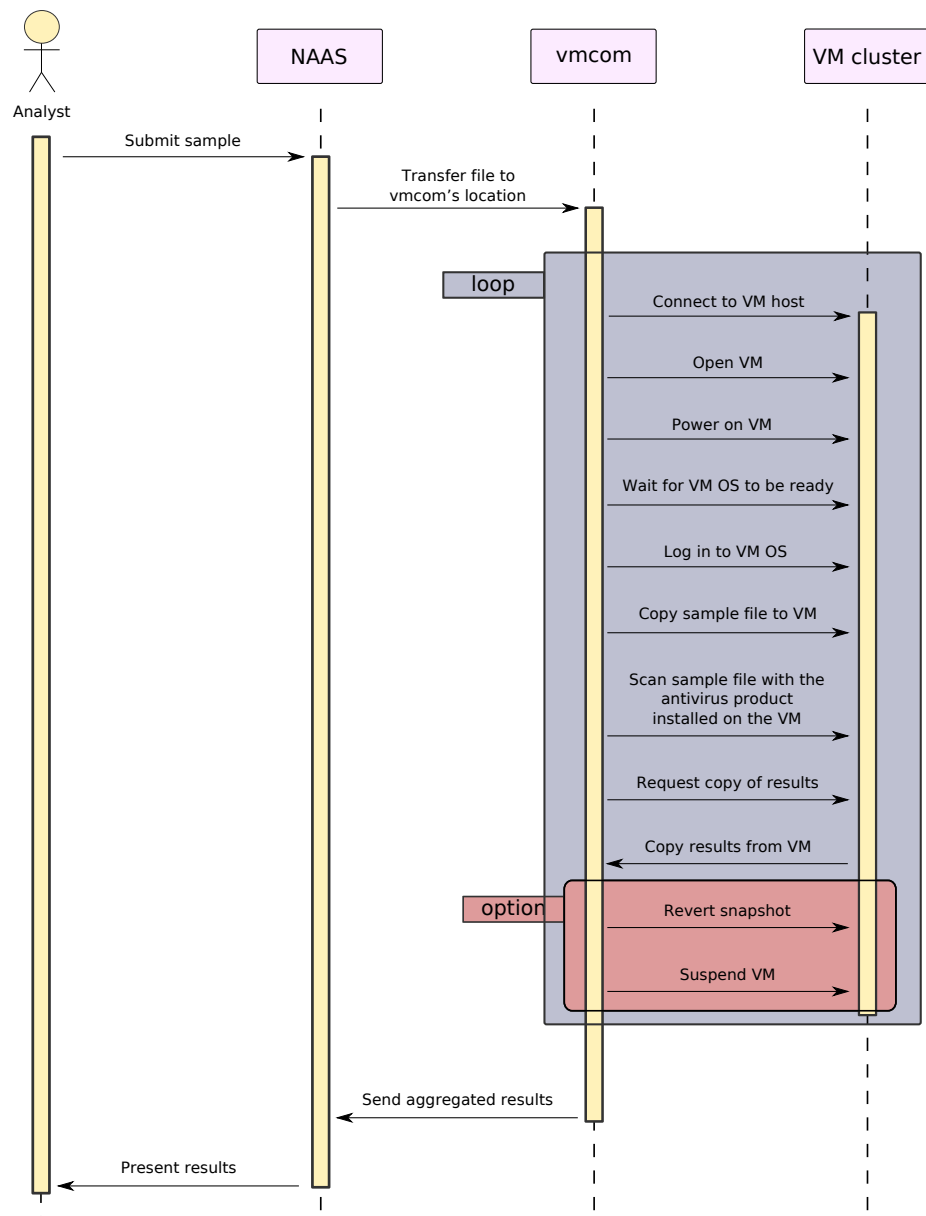


Figure 5.5: Sequence diagram showing interactions for the three first scenarios.

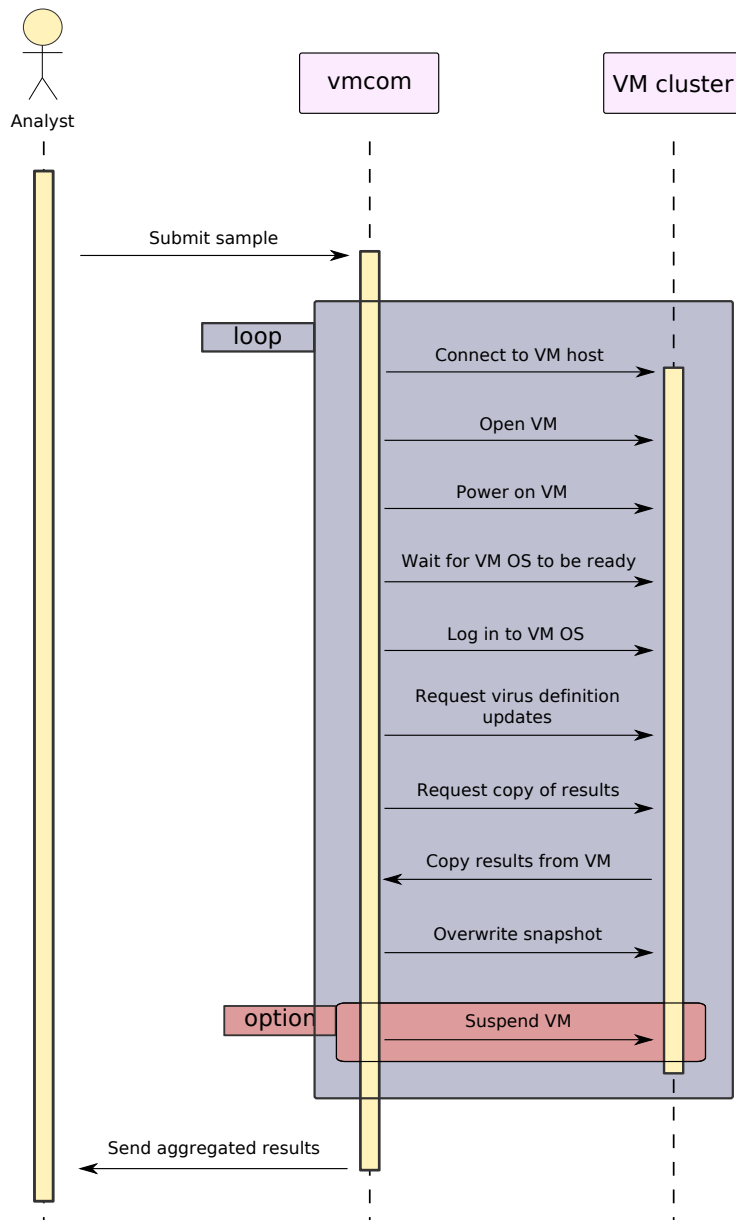


Figure 5.6: Sequence diagram showing interactions for the last scenario.

5.3 Overall list of requirements

The following is the list of requirements elicited from the use case analysis in Section 5.1 on page 43, combined with the high level requirements listed in Section 4.2 on page 32. Early requirements together with functionality introduced from use cases are here collected, refined and structured to give a better understanding of the system-to-be as a whole. Each requirement has its own priority expressing its importance with respect to the others, a brief description and a measurable property. All requirements are shown entirely in Table 5.5 on page 56 and are classified in one of the following two groups.

The F group contains the *functional* requirements for the system, showing its behaviour and results. This group concretise what the system is supposed to do.

The NF group contains the *non-functional* requirements, showing how the system is supposed to be. Contrasting the functional requirements, the NF group does not list specific behaviour.

ID	F.01
Name	Scanning applied by multiple antivirus products
Description	The system shall support an arbitrary number of antivirus products for scanning. A fixed number of products will be supported at the completion of this project. The process shall be automated.
Measurement	File scanning will be done by at least five different antivirus products all of them including usable parsers to aggregate the result files from the antivirus products.
Priority	High

ID	F.02
Name	Presenting results
Description	The system shall be able to write information that is easy to parse to standard output.
Measurement	Observe standard output for results when scanning. Ensure the output is either in XML format or comma separated values (CSV).
Priority	Medium

ID	F.03
Name	Revert to clean states
Description	The system shall be able to restore virtual machines to a clean state upon requested by the analyst. E.g. when the analyst suspects the system might be compromised by malicious code.

Table 5.5: (continued)

Measurement	Each virtual machine has the same state as a clean operating install with one installation of an antivirus product.
Priority	Medium

ID	F.04
Name	Virus definition updates.
Description	The system shall support on demand virus definition updates on all antivirus products supporting this feature with command line.
Measurement	Antivirus products supporting command line updating are updated.
Priority	High

ID	F.05
Name	Remote communication
Description	The system shall be able to call functions for virtual machines located remotely.
Measurement	Virtual machines that are located on different machines than the programs calling them receive commands.
Priority	Medium

ID	F.06
Name	Extending the inventory of products
Description	A possibility of adding new antivirus products shall be available.
Measurement	Integrate a random antivirus product which supports command line scanning and preferably updating.
Priority	High

ID	NF.01
Name	Licenses
Description	The released source code shall be made open source with a suitable license. This excludes all code exposing NAAS design details, as this implementation is kept closed.
Measurement	Source code can be found on the public open source contributing channel http://www.sourceforge.net .
Priority	Low

ID	NF.02
Name	Avoid exposure
Description	No details regarding the NAAS system shall be exposed in any of the produced results during this project.

Table 5.5: (continued)

Measurement	A report and source code evaluation by representatives from NorCERT shall be done before publishing any material.
Priority	High

ID	NF.03
Name	Module based
Description	Following the last requirement requires that the system is in at least two distinct parts and communication between the parts is possible. The closed part may be dependent on the open one, but not vice versa.
Measurement	The open part is made standalone and can operate all its functionality without the closed part.
Priority	Medium

ID	NF.04
Name	Intuitive user interface
Description	The system shall be relatively easy to use
Measurement	An analyst can start using the system in less than two hours of studying documentation (requires that all hardware is ready).
Priority	Low

ID	NF.05
Name	Platform independent
Description	The system shall be independent of specific operating systems.
Measurement	Testing the system on the following operating systems: Ubuntu linux Microsoft Windows XP
Priority	Medium

Table 5.5: Requirements for the implementation part of the project.

5.4 Mapping requirements

The high level requirements defined in Section 4.2 on page 32 are in this section mapped to the requirements defined in Section 5.3 on page 54.

Requirement **HR.01** is covered mainly by **F.01**, but other functional requirements are also partly covering it. **HR.02** is covered by **F.02**, and **HR.03** is covered by **F.06**. **HR.04** is covered by **F.04**, and **HR.05** by **F.05**. **HR.06** does not have a functional requirement since it is hard to measure. Instead,

the performance aspect will be discussed in Section 7.4 on page 73. **HR.07** maps to **NF.01**. **HR.08** is also hard to measure, so no explicit requirements are made. Still, the code will be documented in the source, and a user manual will be written.

Chapter 6

Design

This chapter includes six sections. The first one is about the system architectural design. Starting from the system in general, it describes a high level view of the chosen architectural solution in Section 6.1. Continuing on to Section 6.2, a reasoning for the selection of programming languages can be found. The following three sections, 6.3, 6.4 and 6.5 present a detailed description of the several components that constitute the system. Finally, the last section, Section 6.6 lists the limitations that will arise from the chosen design.

Other than describing the system in terms of components and relations among them, this part of the documentation will register all the significant decisions about the design and the motivations that led to the decisions. It is vital that requirements defined in Chapter 5 can be covered by the design proposed in this chapter, so no design choices conflicting the requirements are made. Doing so will ensure that the design do not prevent the fulfillment of the requirements.

6.1 Description of components

The system-to-be will be split into three different main parts where the first one is the virtual machine cluster, their respective operating system installations and each having one antivirus product installed. This first part may be located remotely from the other two parts, but does not need to. The second part is an adapter program used to communicating with the virtual machines and to perform various operations on them, for instance initialise a file scan. For simplicity and reference, it will henceforth be called *vmcom*, abbreviated from “virtual machine communicator”. Its functionality is listed below.

- Initiate scanning of samples in virtual machines.
- Parse the results from the scans and collect and aggregate them in a proper manner.

- Perform virus definition updates on the virtual machines when requested by the user (on demand).
- Create clean states from the virtual machines and reverting these states if the need to do so emerges. These two functions are done on demand by the user.
- Suspend virtual machines to free hardware memory. Suspending can be applied both automatically and on demand.

The third and last part is the NAAS integration module which will be plugged into the already existing NAAS system. Doing so allow the analysts at NorCERT to continue using NAAS while utilising the system implemented during this project. Initially it could seem unnecessary to split the system like this, but the following important points reasons why it would be done so.

Sharing It is desirable that work from this project stays open and freely available so other people can use the results and possibly contribute to the code. Since NAAS is a closed system and its design details hidden for the public, any code leaking its design cannot be released as open source. Thus by separating and making the main functionality (scanning samples by several antivirus products and so on) independent from all NAAS specific details, the system can still be released to the public without exposing the NAAS design.

Extendability By decreasing the amount of dependencies, one can more easily integrate the main functionality from `vmcom` to other applications.

Portability If the system was to be implemented directly into the NAAS system, the most natural expectation was to use the same programming language as the rest of the NAAS system. When writing the `vmcom` program, the choice of programming language is independent in regards to NAAS. Still, the NAAS integration module shall be able to interact with `vmcom` in some way, so the languages must be selected accordingly.

Virtual machine distribution The cluster of virtual machines will require a significant amount of physical memory, making it difficult to extend the amount of virtual machines when the hardware limits are reached. The `vmcom` program will support distribution of virtual machines to multiple computers, separated from the NAAS system if the cluster grows to those extents. Section 7.4 on page 73 discuss the memory requirement and its problems further.

Figure 6.1 on page 62 graphically displays an overview of the different parts of the planned system by the formal deployment diagram notation [Fow03] from UML 2 [Cob00]. The main functionality from the components is shown in the figure, and the interaction between components and the external actor entity is shown as lines with arrowheads.

Note that the “NAAS interface” component is already existing and implemented, while the “NAAS integration module” will be designed during this project - partly by software developers at NorCERT. The `vmcom` component is the earlier mentioned adapter used to communicate with the virtual machines and perform operations on them (like initiating a file scan by the installed antivirus application). The three different parts explained in this section are marked in the diagram as text with parentheses.

6.2 Choice of programming languages

The clear division of system parts introduces the possibility of using different programming languages where code implementations are needed.

The first part, the virtual machine deployment, does not need any new software implementations.

The second part, merely “`vmcom`”, will consist entirely of implemented code in this project. `vmcom` will be based upon the framework for controlling VMware-based virtual machines, `VIX`. As noted earlier, the `VIX` framework includes wrappers for the languages C, Perl and COM. Additional open source software allows Java and Python programming to use `VIX` as well, but as these two latter projects create an additional level of dependency to the system, they are eliminated as a choice leaving left the candidates C, Perl and COM. COM is a group of proprietary languages which introduce certain unfortunate restrictions like using one particular integrated development environment and operating system for programming. This is a strong negative property and flush away the COM languages as an option without the need for further discussion. Perl and C have both arisen from community work, they have a large user community and does not ship with the same restrictions COM based languages seems to do. Since an important part of the `vmcom` job will be parsing of antivirus log files (scan results), Perl¹ will be preferred as the selected programming language due to its superb text processing capabilities compared to C [Pre00].

The third part, the NAAS integration module, will be written in Python, as the wrapper for including third party tools in NAAS is written in this language. Python is able to run Perl programs, thus able to execute the `vmcom` program. It is also able to read back the results when executing Perl programs.

6.3 Design of the first part, virtual machine deployment

The machine hosting the virtual machines is recommended to be a standalone machine, or at least controlled such that it will not consume all system resources. The virtual machines will be installations of Windows XP. Other operating systems can be used, but Windows XP is most relevant for this project with a memory requirement as low as 256MB.

¹ [SPF08] gives a detailed introduction to Perl as a programming language.

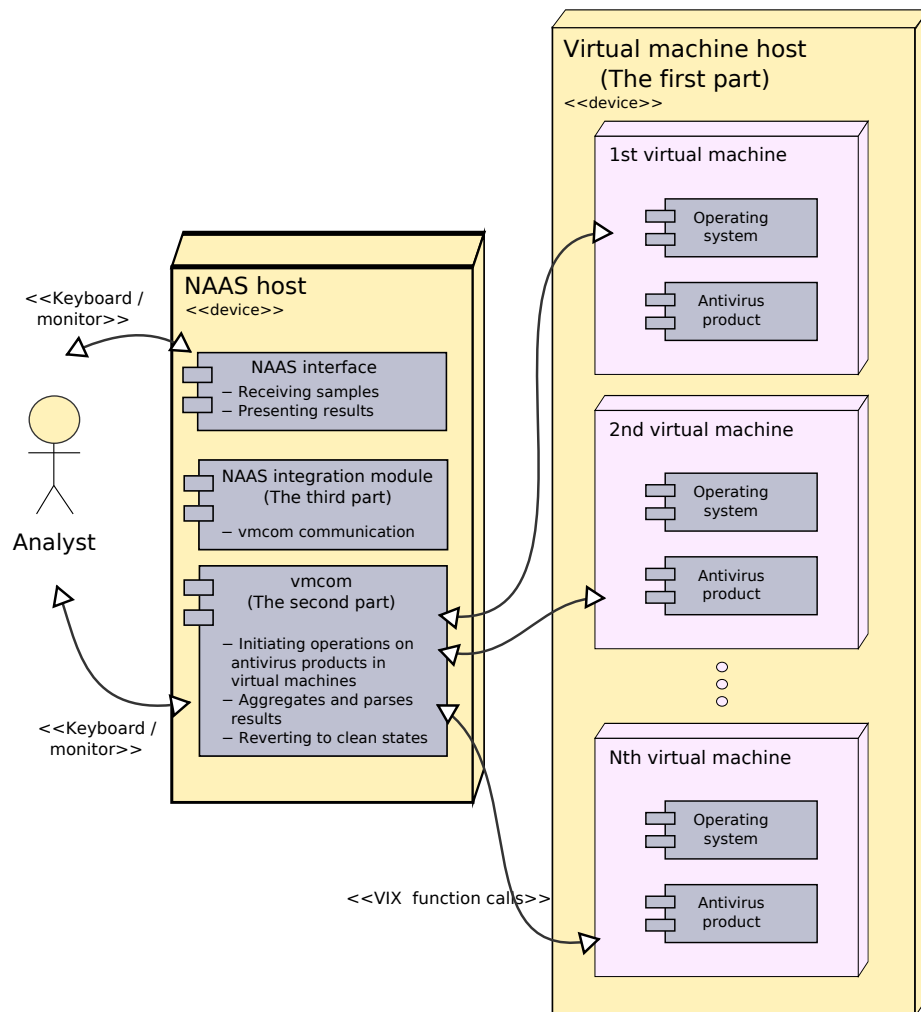


Figure 6.1: Overview of system parts and their interactions in the deployment diagram notation [Fow03]. The three different parts are marked in the diagram (“Virtual machine host”, “vmcom” and “NAAS integration module”).

For a machine with 1GB of available memory, there is obviously not space for plenty of concurrent virtual machines. If we assume that the host operating system also requires 256MB of memory, there is only space for three virtual machines as shown in Equation (6.1). This is in theory, and the chance that the host operating system has such a small amount of memory usage is slim.

$$\begin{aligned}
 \text{Available memory} &= \text{Host OS memory} + \\
 &\quad ((\text{Number of virtual OS}) \times \\
 &\quad (\text{Minimum memory for each virtual OS})) \\
 1024 &= 256 + (X \times 256) \\
 X &= 3
 \end{aligned} \tag{6.1}$$

When the memory limit is reached, the system will start to swap memory to disk, and the computer will spend too much time shuffling data back and forth from disk and memory (*thrashing*) [Den68], leading the system to be basically useless. Due to this phenomena, it is essential that the system has enough memory at all times. Actually, it should be packed with as much memory possible to work optimally. Section 7.4 on page 73 covers a more throughout discussion regarding this topic.

The selection of antivirus applications to support the system-to-be is chosen primarily on their licenses. The antivirus products free of charge are prioritised prior to the rest of the products, with the additional requirement that they support command line scanning of files. When the available free products are integrated in the system, trial based versions of cost licensed products will be added as long as there is time available in the project. Table 6.1 on page 68 shows different costs for a selection of products. Most of the free products are only so when used in a non-commercial setting, and require a licence elsewhere. There is no need to go into juridical details here, so the free editions can be, and probably will be, used to create the parsers whether or not the actual product will be a commercial edition or free home user edition. It is assumed that the result files generated by the different editions can be parsed in the same way if a commercial edition is to be used. Each virtual machine will also require its own Windows XP license.

6.4 Design of the second part, the vmcom program

`vmcom` will be coded in Perl, and will control and handle all virtual machines set up in a configuration part of its program body. Most of the functions for this part is already mentioned in a general setting, but is below refined to give an overview.

Scan are the basic use and most important functionality of `vmcom`, and will be used to scan samples for viruses.

Take snapshots will be used to store a clean system state.

Revert snapshot will be used to go back to a clean system state from a previously stored state.

Update will be used to update virus definitions to ensure an up-to-date result from the different products.

Suspend virtual machines will be used where appropriate to free system memory.

Power off virtual machines is a functionality made available to use when for some reasons virtual machines need a hard restart.

An example of implementing the scan-functionality is given as a pseudocode snippet shown in Listing 6.1. Since similar code structures will be used for the rest of the functionality so no more pseudocode examples are considered to be needed to get the overview. A example of the actual implemented code can be found in Section 7.2 on page 69.

```

1 VMHandle ← ConnectToVirtualMachine("VirtualMachineImage.vmx")
2 ;
3 PowerOn(VMHandle);
4
5 while(VMHandle ≠ ready) {
6     wait;
7 }
8
9 AuthenticateUserInVirtualMachineOS(VMHandle,
10     "username",
11     "password");
12
13 CopySampleFileFromHostOS(VMHandle,
14     "source_file",
15     "destination_file");
16
17 RunProgram(VMHandle, "c:\program.exe");
18
19 CopyResultFileToHostOS(VMHandle,
20     "result_file_source",
21     "result_file_destination");
22
23 ParseResultFileFromHostOS("result_file_location");

```

Listing 6.1: Pseudocode for on demand automatic scanning of one sample file.

As the memory consumption from the virtual machines can extend the amount of memory currently possible to fit in a single-host system, the `vmcom` program will be designed to allow suspending virtual machines. This will free up used memory, making space for new virtual machines. However, powering on the

virtual machines is required prior to initiate an operation on them when they are in a suspended state. Powering on the virtual machines require a large amount of time, and might cause long delays.

The `vmcom` program will support distributing the virtual machines to different host machines, making work balancing a possibility.

6.5 Design of the third part, the NAAS integration module

A tool integration framework to NAAS is under development by software developers at NorCERT. To avoid leaking NorCERT's available processing power, neither NAAS nor the tool integration can be described in detail in this report. However for this project, it is sufficient to give an overview of the design and how both NAAS and the tool integration works. This will be enough for the reader to understand NAAS and its tool integration functionality in general.

NAAS will support third party applications when the integration framework is complete, and integrate these applications in such a way that they can be executed through the normal NAAS user interface. The integrated applications are command line based for simplicity, and should preferably be coded as standalone as possible. This means basically that the program should be able to run without NAAS, and use NAAS as a graphical user interface component only. For setting up integration to a standalone external application, which in this example is the `vmcom` program, only a few settings in a Python class is needed. To understand this, consider Figure 6.2 on the next page. The figure shows how third party applications can be connected through NAAS by using the framework from NorCERT. The framework must be used when implementing the NAAS integration module. The figure displays the different operations (called "tools"), collected in a container ("ToolContainer") which is instantiated by an "AnalysisClient" from the NAAS application. To execute one of the operations `vmcom` provides, simply run the corresponding tool, for example "`vmcomScanningTool`" which is the operation for scanning a file. After the tool's execution, the AnalysisClient will receive the results in a textual form and further deliver it to the NAAS system for representation, and to persistently store the result data. The framework are under development at NorCERT, while all the `vmcom` "tools" together form the NAAS integration module of this project.

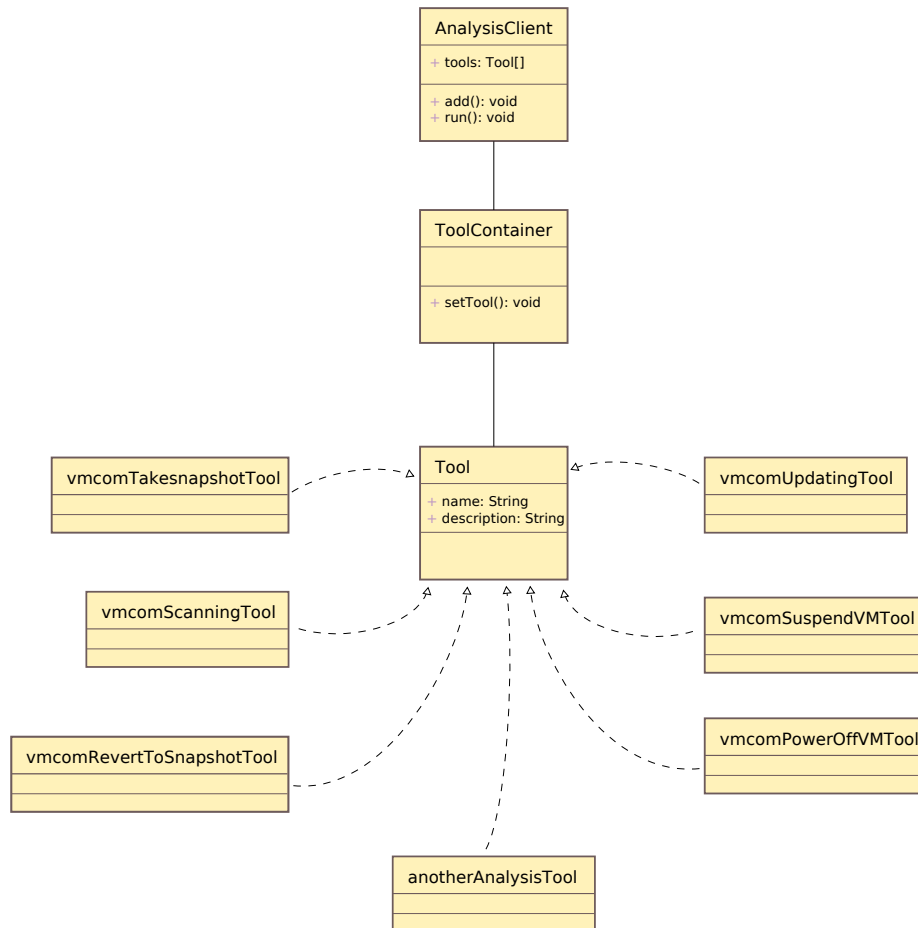


Figure 6.2: The NAAS integration displays how different third party “tools” can connect to the main NAAS system. Each tool has its own operation, for example does the “vmcomScanningTool” initiate a file scan by using `vmcom` through NAAS.

6.6 Design limitations

Since the `vmcom` program will communicate with the antivirus application on the virtual machine based on command line programs, antivirus products requiring interaction through a graphical user interface cannot be integrated in this system. According to `VirusTotal`'s webpage, their solution is based on command line tools, just as this project will be. This indicates that this limitation will probably not be that big of a problem. Results from the `PowerScan` thesis [LL08] shows that the authors performed the tedious work of eliminating products without command line interfaces, saving this project hours of effort of doing the same work. Table 6.1 on the following page is based on their results and shows that this limitation does not affect a large percentage of antivirus products. Some of the values in their table was not correct and is hereby changed, and there are also some additions to the list of products. At the aggregation of results from the operations, the result files will be transferred to `vmcom` for parsing. This requires the antivirus products to be able to write results to a log file. Since the programs will run in command line, the batch scripting available in Windows supports writing the command line output to a file.

Product	Command line features		
	Scan	Update	Free of charge?
Avast! Home Edition	✓	✓	✓
AVG Antivirus Free Edition	✓	✓	✓
Avira AntiVir Personal Edition	✓	✓	✓
BitDefender Free Edition	✓	✓	✓
BullGuard Internet Security	✓	✓	⊗
CA AntiVirus 2008	✓	✓	⊗
ClamWin Antivirus	✓	✓	✓
Comodo Antivirus Beta version	✓	⊗	✓
ESET NOD32	✓	⊗	⊗
F-prot	✓	⊗	⊗
F-Secure Antivirus 2008	✓	⊗	⊗
Kaspersky Antivirus	✓	✓	⊗
McAfee Virus Scan	⊗	⊗	⊗
McAfee Virus Scan Command Line Client	✓	⊗	⊗
Norman Antivirus	✓	✓	⊗
Norton Antivirus 2008	⊗	⊗	⊗
Panda Antivirus 2008	✓	✓	⊗
Sophos Antivirus SBE	✓	✓	⊗
Trend Micro 2008	✓	⊗	⊗
ZoneAlarm Antivirus	⊗	⊗	⊗

Table 6.1: On-demand features in, and cost of a list of different antivirus products. The content is based on data from the PowerScan thesis.

Chapter 7

Implementation

This chapter contains a description of how the system is realised. The three first sections, 7.1, 7.2 and 7.3 are split in a similar way as in the previous chapter where each part is written separate from the other two. Section 7.4 discusses the performance aspect of the implementation, and gives guidelines to get the best possible performance from the system.

7.1 Implementing the first part, Virtual machine deployment

Each antivirus product requires its own virtual machine with its own operating system. Installing this is trivial, but required. How to install and setup the system are covered in Appendix A. The selection of antivirus products supported by `vmcom` at the end of this project is shown in Table 7.1 on the next page. The two latter columns indicates if functions are successfully written to automatically run scanning of files and virus definition updates, respectively.

7.2 Implementing the second part, the `vmcom` program

The `vmcom` program is written in Perl, and applies operations on one or all virtual machines on demand. The program supports command line arguments to control its behaviour. As an example, the argument `scan /file/path` will initiate virus scan on the given file path. If no arguments are given, a brief help screen will be printed to the user. Using arguments is done to more easily switch between the available operations without the need of any modification to the source code. The program consist of a configuration part, where necessary information for *each* active virtual machine must be set prior to using the antivirus product installed on the current virtual machine. Settings are as follows.

Product name	Scan	Update	Comment on update
F-Secure AntiVirus	✓	⊗	Deprecated on demand functionality
AVG Free	✓	✓	No explicit textual results
BitDefender Free	✓	✓	
Avast Professional	✓	⊗	Not supported
Kaspersky Anti-Virus	✓	⊗	Script do not download files to the correct location
Panda Engine 1.4.3	✓	⊗	Not supported
Trend Micro	✓	⊗	Not supported
McAfee	✓	⊗	Not supported
ESET NOD32	✓	⊗	Not supported
Avira AntiVir	✓	✓	Update process exits as a background process. Works if system does not suspend.
ClamWin	✓	✓	

Table 7.1: Antivirus products and their on demand features supported by the implemented solution. The last column comments on each non-trivial case of on demand update.

- The name of the antivirus product
- Information for the machine hosting the corresponding virtual machine
- Path to the antivirus executable together with arguments for running a command line scan
- Path to the antivirus update executable and arguments for running a command line update
- Path to the corresponding virtual machine image (File with “.vmx” suffix)

Operations on the virtual machines are needed various places in the program, and to avoid duplication of code, important operations are made as *subroutines*. Subroutines are Perl’s way of creating a Java method or C function. Consider the operation “copy a file from host operating system to the guest virtual machine”. As the operation is needed by both the update and scan operations, it is made as a subroutine and shown in Listing 7.1 on the facing page. Prior to a file copy, other subroutines must be called and must run successfully. Examples are connecting to the virtual machine, and logging in as a local user. If one of the subroutines for an antivirus product receives an error, *vmcom* will abort the running of the *current* product and run cleanup processes depending on when the error was reported. A failure will only halt the current running antivirus product, and continue to the next product in line. An activity diagram following the UML syntax [Fow03] has been drawn to show the explained program flow. The diagram shows prerequisite steps to the different operations available in *vmcom*, and can be found in Appendix B.

```

1 sub copy_file_to_guest {
2   # Initialise variables
3   my($err, $vmHandle, $subject_file_location) = @_;
4
5   # Run VIX command and store return value $err =
6   VMCopyFileFromHostToGuest($vmHandle, $subject_file_location,
7     # src name
8     "c:\\scanme", # dest name 0, # options VIX_INVALID_HANDLE); #
9     propertyListHandle
10
11  # Abort if an error is found and suspend the virtual machine
12  so memory will be freed
13  abort("VMCopyFileFromHostToGuest() failed", $err) if $err !=
14    VIX_OK;
15 }

```

Listing 7.1: Code example showing subroutine for copying a file from the host operating system to a guest virtual machine.

`vmcom` is running a threaded model in its core to allow concurrent operations on multiple virtual machines. A system with a single processor will not benefit from the threads, but if there is more than one processor available (multi core architecture or a distributed cluster), concurrency can be enabled. There are some issues regarding Perl's native thread models, notably the fact that enabling the functionality requires a non-default switch prior to compiling the program language binaries. Luckily, there exists a module that use the same API as the built-in threads in Perl. The module is called “forks”, and is utilised in `vmcom` to avoid the need for recompiling Perl. The forks module is used such that it will not introduce any concurrency issues like deadlocks [CES71] or starvation [Tan01].

For the scanning operations, the results are immediately returned to the host operating system when the scan is complete. `vmcom` will then aggregate the result by parsing the log file and report to the user. As the syntax used in each result log file is unique, the parsing mechanism varies for each antivirus product. A parser part example for the product `ClamWin` is shown in Listing 7.2 to give an overview. Complete source code can be found in Appendix B.

```

1 sub parse_results { # Initialise variables
2
3   my $results = undef; my($type, $av_product,
4     $result_file_destination) = @_;
5
6   # Open the result file and read it into the data array.
7   Close when done
8   open(RESULTS, $result_file_destination) || die "Could not
9     open file!";
10  my @data = <RESULTS>; close(RESULTS);
11 }

```

```

9      # A normal case-switch used to separate the antivirus
      products.
10     # (Only one case is shown in this example)
11
12     switch ($av_product) {
13
14     case "ClamWin" {
15         # Splitting the result line on the characters "; "
16         my ($text, $data) = split(/: /, $data[0]);
17         # Remove carriage returns
18         chomp($data);
19
20         # If the text matches "OK" the file is clean
21         if ($data =~ /OK/) {
22             $results = "clean";
23         }
24         else {
25             #Otherwise it is not. Virus information is stored in the
              $data scalar
26             $results = $data;
27         }
28     }
29 }
30

```

Listing 7.2: Example parser code for one of the antivirus products.

After the completion of parsing each result file, the user will be notified whether the file was found clean or not. The aggregated results are printed in comma separated field at the process completion.

7.3 Implementing the third part, the NAAS integration module

The NAAS integration module consists of a set of “tools”, as explained in Section 6.5 on page 65. The complexity level of integrating *vmcom* into NAAS is low, as *vmcom* is written completely standalone. The integration framework is not finished from NorCERT’s part, so the code shown is not the finished version of the integration module. The code is using the current interfaces given by NorCERT, but is only shown to display how little effort is needed to integrate a third party tool into NAAS. Listing 7.3 shows the required Python code to integrate the scanning functionality from *vmcom* into NorCERT’s integration framework as the framework is currently designed.

```

1  # Import the generic NAAS integration module.
2  # (The integration module class is under development by
      NorCERT, and

```



```

3  # therefore not included here)
4  from naas import *
5
6  # vmcom scanning tool class
7  # (The AnalysisTool class is under development by NorCERT, and
8  # therefore not included here)
9  class vmcomScanningTool(AnalysisTool):
10
11     # Class Constructor:
12     def __init__(self):
13         # Set own name:
14         self.name = 'vmcom scan'
15         # Set location for script
16         self.script_location = '/home/km/vmcom.pl'
17         # Assign arguments to the script
18         self.arguments = '--scan ' + filename
19         # Initialise parent:
20         AnalysisTool.__init__(self)
21         # Run the tool
22         self.run
23
24     # Run function for vmcom scan
25     def run(self):
26         # To be implemented by NorCERT through the
27         # AnalysisTool class
28         return True

```

Listing 7.3: Python example code showing the necessary code to integrate `vmcom` into the NAAS framework.

7.4 Performance

After the submission of a sample file, the system should respond in a *reasonable time* as mentioned in high level requirement **HR.06** found in Section 4.2 on page 32. The time requirement do not have an exact value in seconds, but needless to say, a fast system is more valuable than a slow one. This section covers the aspects of performance by the system, and how to obtain a lowest possible running time from the system.

7.4.1 Running times

As mentioned in Section 6.3 on page 61, the amount of memory is a critical factor for keeping the time to perform a job request to a minimum, processing capabilities on the computer should be prioritised secondarily. An example of a file scan will be used to show this, but the same applies for the rest of the main functionality from `vmcom`. Prior to scanning a file, the virtual machine must receive the file. This file is copied from the host operating system, but is

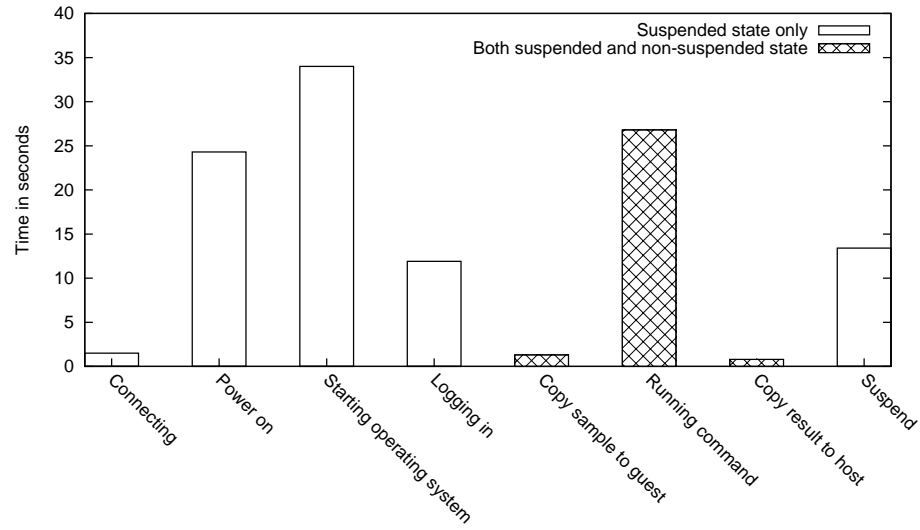


Figure 7.1: Running times for each operation on the virtual machine.

a trivial task regarding amount of seconds¹. However, for the file copying to be possible, the virtual machine must be powered on, the operating system must be started and a registered user in the virtual operating system must be logged in. This requires a substantially delay compared to copying the file. The required time for scanning a file varies from the different antivirus products, but the rest of the operations use more or less the same time to complete independently of the antivirus product. To free memory, the virtual machine is suspended after completion of the scanning, and is also a time demanding operation. Figure 7.1 is showing this as a boxed diagram with the average time each operation took for nine antivirus products. The patterned filled bars are showing operations that are required for all virtual machines independently of which state they are in (“on”, “off”, “suspended” or “logged in”). The other bars show delays when virtual machines are suspended after completion, and the next in line have to be filled into memory prior to the scan. As the figure indicates, this consumes a large percentage of the total time and should be avoided to keep the running times as low as possible.

7.4.2 Resource utilisation

Whether or not to use a threaded model depends upon the utilisation of the processor (CPU). Say as an example that only 30% of the CPU was in use when

¹ Applies when the file is small (< 200KB). The delay increases when the file size increases.

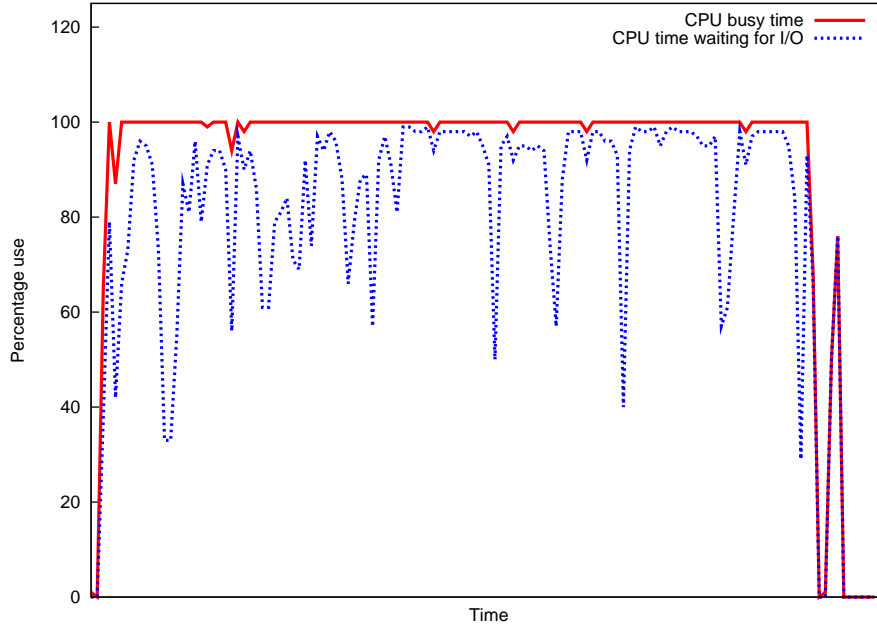


Figure 7.2: Total utilisation of CPU during scanning of a file. The figure shows that the CPU is never idle during the scan, and has around 100% CPU utilisation constantly.

running a file scan. In theory, three antivirus products could run concurrently and utilise $30\% \times 3 = 90\%$ of the CPU. On the test system, this was not relevant as the system constantly used 100% of the CPU as shown in Figure 7.2. Using Equation (7.1) as CPU time definition, the system was never idle during the scan but mostly waiting for input/output operations from the disk (I/O). Even if this test system would not gain anything by concurrent operations, it is important to notice that if the system was a multi-core architecture, these numbers might be different as each CPU could work on its own thread.

$$\begin{aligned}
 100\% &= (\text{Kernel mode CPU time})\% \\
 &+ (\text{User mode CPU time})\% \\
 &+ (\text{CPU time waiting for I/O})\% \\
 &+ (\text{CPU idle time})\%
 \end{aligned} \tag{7.1}$$

Memory usage during a file scan was reasonably high and the available memory in the test system was filled almost instantly. Figure 7.3 on the following page graphically shows this. There is only around 25000 bytes of free memory, and indicates that the system was near an upper limit. The constant line is showing the swap usage, indicating that the operation does not affect the swapping of memory. If however the memory limit was reached, the swap usage would

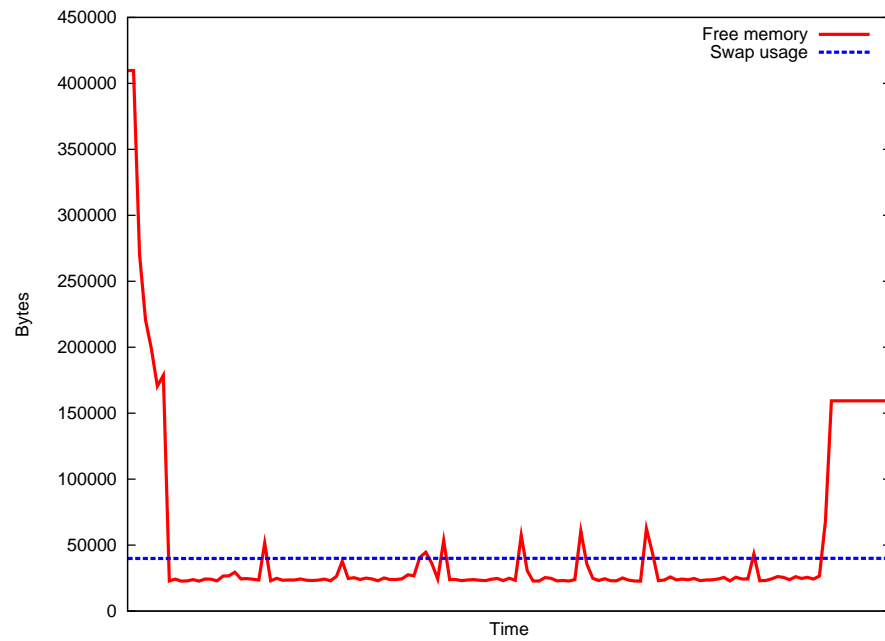


Figure 7.3: Total utilisation of system memory during scanning of a file. Free memory is kept to a minimum on the system used, indicating the system memory limit was almost reached.

increase [ARS89].

Chapter 8

Project evaluation

This chapter contains an evaluation of the main parts of the project. Section 8.1 discusses which of the goals defined were fulfilled or not, and gives a brief explanation if a goal was unsuccessful. Section 8.2 contains an evaluation of the involved parties during this project, and continues with Section 8.3 that discusses further work. The chapter ends with Section 8.4 which contains a discussion about the most important challenges during the project.

8.1 Fulfilment of project goals and system requirements

Each of the goals defined in this project are evaluated in this section. The goals derive mainly from fulfilling the requirements stated in Chapter 5, but also the result goals are considered. A summarised table is shown in Table 8.1 on page 79.

8.1.1 Result goals

The task given is considered to be accomplished by the implemented system and the delivered report. This assumption is primarily based on achievements of the result goals stated in Section 1.2 on page 6. Result goal **RG.01** is covered by material in Chapter 2 and Chapter 3, while **RG.02** is covered by studies documented in Chapter 4. The implemented software results in the accomplishment of **RG.03** and **RG.04**. All deliverances (software and this report) are distributed freely and available for everyone, thus covering **RG.05**.

8.1.2 Software requirements

Fulfilment of the requirements listed in Section 5.3 on page 54 are in this section each evaluated from top to bottom, starting with the functional requirements.

Requirement **F.01**, which is one of the main software requirements, is fulfilled with a good margin. The `vmcom` software supports file scan and result parsing by eleven antivirus products, thus exceeding the required amount with 220%.

Requirement **F.02** is satisfied by the use of a comma separated result output. Further, requirement **F.03** is fulfilled by supporting both saving system states and reverting to these states on demand.

The next requirement, **F.04**, is only partly fulfilled. Most of the antivirus applications support command line scanning, but few of them support command line updating since their design are based on an automatic updating routine. The applications integrated in `vmcom` that support command line updating will be updated on demand. There are some products that do not support command line updating, but might still work by utilising some custom workarounds as mentioned as further work in Section 8.3 on page 80. Not all of the antivirus applications will for this reason be updated by `vmcom`, and must run active in the background (non-suspended) to automatically receive the updates. See Table 7.1 on page 70 for an overview over products supporting on demand updates.

Requirement **F.05** is fulfilled by using the VIX-API for remote communications.

Requirement **F.06** is also fulfilled, with instructions explained in Appendix A.

The first non-functional requirement, **NF.01** is a license based one. It is marked with low importance, and this is a correct choice when comparing to the rest of the requirements. However, having an open source license makes this project unique compared to already existing similar solutions, and makes the requirement important for the project. In either way, the requirement is fulfilled as the entire report is published together with the `vmcom` software. Code exposing NAAS will not be published since NAAS design details should not be exposed leading to fulfillment of requirement **NF.02**. This statement was approved by representatives from NorCERT before delivery of the report and software.

Requirement **NF.03** is fulfilled by separating `vmcom` such that any other implementation can utilise its power. Requirement **NF.04** is one of the hardest ones to measure, but with the relatively easy to use interface together with a brief help page and detailed user manual, this requirement is considered fulfilled.

As `vmcom` is written in the interpreted language Perl it is platform independent, thus fulfilling requirement **NF.05**.

8.2 Involved parties

There are three different parties connected to this project. Firstly, there is NTNU as the academic side. Secondly, NorCERT is the external part suggesting the scope for the project. The third part is me, struggling to ensure that the produced results are in according to the scope for the project and in a good quality at delivery.

Goal	Comment
RG.01	Accomplished by delivery of Chapter 2 and Chapter 3.
RG.02	Accomplished by delivery of Chapter 4.
RG.03	Accomplished by delivery of the implemented software.
RG.04	Accomplished by delivery of the software.
RG.05	Accomplished by the delivery of the report and the software together.

(a) Result goals

Requirement	Comment
F.01	Fulfilled by the <code>scan</code> functionality in the <code>vmcom</code> program.
F.02	Fulfilled by a comma separated text based output from <code>vmcom</code> .
F.03	Fulfilled by the <code>takesnapshot</code> and <code>revert</code> functionality in the <code>vmcom</code> program.
F.04	Partially fulfilled due to antivirus products lack of supporting command line updating.
F.05	Fulfilled since <code>VIX</code> supports remote communication.
F.06	Fulfilled and explained in Appendix A.

(b) Functional requirements

Requirement	Comment
NF.01	Fulfilled as <code>vmcom</code> and the report is freely available.
NF.02	Fulfilled as the design details of NAAS remain hidden.
NF.03	Fulfilled as system parts are separated from each other and module based.
NF.04	Considered fulfilled by the report and code documentation.
NF.05	Fulfilled by using Perl to implement <code>vmcom</code> .

(c) Non-functional requirements

Table 8.1: Summarised list of the accomplishment of result goals and stated functional and non-functional requirements.

8.2.1 A threesome satisfaction?

A traditional project written in cooperation with the university alone does have academic requirements, such as a innovative idea and a well written report and supporting documentation. In a project with an external part like this one, this of course still applies, but there are also some additional aspects to consider. The external part may be more focused on a properly implemented and usable solution than a report. This project has proven to have a good balance between a good report including studies regarding the possibilities of implementing the requested system, and also the actual realisation of it in form of Perl and Python code. Needless to say, for this project to be considered successful, both the university and the external part have to be satisfied. Satisfying these parts are not necessarily trivial, but there are currently no indications that any of them are dissatisfied.

8.2.2 Meetings and communication

During the work period there were four meetings. The first one was held with all parts present at NorCERT's location. Some guidelines for the project were set, and the final text regarding the problem description was determined. Since the text was relatively extensive for a project consisting of 15 credits, all parts agreed that the opportunity of later on reducing the scope was present. However, the text has not been changed. The problem description is written entirely in Section 1.1 on page 5.

The next meeting was in Trondheim with Skramstad with the agenda of general guidelines of structuring the report.

Meeting number three was held again at NorCERT's location, this time with representatives from NorCERT. The agenda was to sketch a requirement specification for the NAAS integration design, that later on could be used for the `vmcom` integration with NAAS. The results can be read in Section 6.5.

The following meeting was held with Skramstad, and valuable help was given as he had proofread the entire report.

A final meeting is scheduled with NorCERT after the delivery of the report. NorCERT will receive a copy of the report, and the implemented system will be presented.

8.3 Further work

There are always ways to improve a product, and the system implemented in this project is not an exception. This section covers some of the areas that could benefit from further work.

8.3.1 Improve on demand update functionality

The requirement only partly fulfilled in this project is one of the areas that could be improved. The system lacks on demand updating functionality for several of

the antivirus products integrated. The reason for this is mentioned in Section 8.4, but is mostly because that the products did not include this functionality at the same level as scanning files. Some workarounds exist and were tested but would introduce unnecessary complexity to the project if included. Additionally, there were some issues with all of the workarounds so none of them were included in the delivery. As an example, a batch script could be made for the McAfee product that used the date today and downloaded virus definitions from a FTP site which organised definition updates in folders named todays date. After download, the files could be unpacked and replace the existing ones on the system. A batch script in Windows has limited functionality, and even getting the current date is not straightforward. Of course, the download of the definition files could be applied on the host Linux system and transferred to the McAfee virtual machine easily, but would require a redesign of `vmcom` as it is now.

8.3.2 Multiple antivirus products on a Linux based virtual machine

As explained in Section 4.1.1, there is difficult to install multiple antivirus products on the same system. However, this applies only to the Windows operating systems, since Linux based antivirus products do not tend to hook themselves to the API. By utilising this fact, there is possible to use all Linux based antivirus products on one virtual machine, thus decreasing the necessary total number of virtual machines. There might be issues with such an approach, but the time delay could be dramatically reduced if the approach works.

8.3.3 Structuring the Perl code

The `vmcom` program grew continually, and the code readability could be better. A more readable program structure is therefore suggested, and moving the configuration elements out the main program is one of the possibilities of doing so.

8.3.4 VMware process cleanup

If `vmcom` exits abnormally, there is a chance that virtual machine processes do not exit as they should and remains running in the background. A new improved version of `vmcom` could check for such processes, and clean them up to avoid consuming system resources.

8.3.5 A testing framework

A program is not complete until it has a reasonable test coverage. Automated unit- and functional tests should be made and continuously executed to ensure all functionality works as it should.

8.3.6 Improvements beyond vmcom

The process of dynamic analysis is huge and more tasks could be automated, or at least made more efficient. Running suspicious software and observing the results is one of the areas that could be studied, and the results could be implemented as a similar module to `vmcom`.

8.4 Challenges

Some difficulties emerged during the project, and the most important of them are described in this section. The challenges without a feasible solution have been frustrating, but most of the challenges have been manageable.

8.4.1 VIX - pros and cons

The programming API was indeed a prerequisite for this project to succeed, as implementing such an API would require considerably more time than available. Still, being dependent upon such an API is not always optimal, especially when it is so high level that you it does not give too much information if anything goes as unexpected. Luckily, this has not crippled the implemented solution, and VIX has proven itself as a good and pretty reliable API. There are some exceptions, notably that the system may hang in special circumstances, but this is solved by aborting the execution by the timeout functionality implemented in `vmcom`.

8.4.2 Slow hardware

The machine(s) used to do tasks described in this report is meant to be powerful and top-notch. This is particular so for the amount of memory, as each virtual machine consumes a significant amount of memory. The machine used in this project was of the rather old type with a small amount of memory and a disk with only 40GB. This slowed down testing the functionality and made handling virtual images hard, as each of them required up to 5GB of disk space each when snapshots were included.

8.4.3 Command line update functionality missing

As most of the antivirus products includes a reasonable good command line scanning tool, only few of them have the same functionality for the virus definition updates. This was a tricky task to solve, and in some cases too hard to manage, or even practically impossible. For the ones without the functionality out of the box, workarounds were found with assistance from various internet sources like forum posts describing a similar problem. This required a reasonable amount of time to get right and required a cumbersome try-and-fail process for each relevant product.

The update functionality when using the mentioned workarounds, did only work partially. For this reason, the antivirus products requiring specific workarounds are currently not supported by `vmcom`. Implementing and testing the workarounds required too much time and effort, and was not prioritised. Systems using `vmcom` in production mode will most certainly be packed with enough memory to keep the virtual machines powered on at all times, making the on demand update functionality unnecessary.

8.4.4 Switching VMware software

Initially, the project started out using the workstation edition of **VMware**. Later on when the server edition proved as a better alternative for the task, a virtual machine image migration had to happen. To manage this, a couple of settings in the images metadata had to be changed accordingly. As this was an undocumented hack from **VMware**'s side, there was a possibility of introducing unwanted side effects. Luckily, no indication of such behaviour was seen.

8.4.5 Segmentation faults

The initial combination of the Perl installation, operating system and **VMware Server** introduced a shared memory software issue when running the `vmcom` program. A swift debug process found problems in the drop-in replacement module "forks" that is used to enable a threaded module as explained in Section 7.2 on page 69. The problem was not tracked down, but an operating system upgrade together with a new version of Perl, **VMware Server** and the VIX-API magically solved the problem with segmentation faults.

Chapter 9

Project conclusion

Malicious software are designed more and more sophisticated and intelligent. The threat from malicious software becomes imminent and dangerous when combining the levels of sophistication with the increasing number of discoveries of such software. People with malicious intent, whether it is economical, warfare or any other reason, can construct software to reach their evil goals. The process of analysing and understanding the functionality from potential malicious software is complex and time consuming. Thus to be able to cope with the large amount of threatening software, there is a need for tools to make the analysis process as swift and easy as possible. Antivirus products are often used in an analysis, especially at an early stage, to get an indication of the functionality of the malware, its uniqueness, and its level of threat. One of the efforts to ease the analysis process is the automation of the virus scans, and was the main goal to arrange in this project. In cooperation with representatives at NorCERT, requirements for a system was sketched and a full realisation of an automated virus scanning system was realised. Similar solutions already exist, but the results from this project is a configurable, freely available and open source system. These are properties making the project unique, where everyone can use the results in their systems and even contribute to the released software implementation if they want to do so. A key requirement for the system, was to build it module based. By using a module based architecture, the produced system can easily be plugged into existing solutions which gives the possibility of reuse. The implemented system is based on existing freely available solutions from VMware, which made the relative extensive problem description possible to solve in the time given.

All result goals defined in this project are accomplished, and practically all of the sketched requirements are fulfilled. The functionality from the requirement only partly fulfilled is considered further work, and a temporary solution is explained in the report. The system has been continuously run through the project and has proven to operate like requested. The system is ready to be deployed, and the source code and user manual together can be used to set it in production when NorCERT's software developers complete their NAAS

integration framework.

Antivirus vendors are struggling a hard fight against the current software threats, and there is possible that the legitimate part will loose the battle in the end. Malware easily outsmarts antivirus products by utilising clever tricks, and new technology might be needed to avoid the threats. Exactly how futuristic applications will work is uncertain, but there is a chance that a more automated dynamic analysis could deduce functionality from software better than the signature based approach which is done today. It is also possible that signature based detection will be outdated and abandoned if there will be a way to understand the syntactic content of a sample by utilising the powers from dynamic analysis methods. If this automated dynamic analysis could use a generic method to study a sample's intentions with the same cost as antivirus applications today, obfuscating techniques or other common tricks malware use to avoid signature detection will probably no longer be a problem. Until then, antivirus products can be used to scan samples for threats, and the system implemented in this project is a possible solution to easily scan for threats by automating the use of multiple antivirus products on the same data and aggregate the results in a proper manner.

References

- [AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.
- [Ahm07] David Ahmad. The contemporary software security landscape. *IEEE Security and Privacy*, 5(3):75–77, 2007.
- [ARS89] E. Abrossimov, M. Rozier, and M. Shapiro. Generic virtual memory management for operating system kernels. *ACM Symposium on Operating Systems Principles*, pages 123–136, 1989.
- [av-08] AV-Test release latest results, 2008. http://www.virusbtn.com/news/2008/09_02 Last date accessed 2008-10-01.
- [BC94] Steven M. Bellovin and William R. Cheswick. Network firewalls. *Communications Magazine, IEEE*, 34:50–57, 1994.
- [Bra97] S. Bradner. RFC 2119: Key words for use in RFCs to indicate requirement levels, March 1997. Status: BEST CURRENT PRACTICE.
- [Bra00] Eric Braude. *Software Engineering: An Object-Oriented Perspective*. Wiley, 2000.
- [CES71] E.G. Coffman, M.J. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 1971.
- [CGME07] Jamonn Campbell, Nathan Greenauer, Kristin Macaluso, and Christian End. Unrealistic optimism in internet events. *Computers in Human Behavior*, pages 1273–1284, 2007.
- [Cob00] Cris Cobryn. Modeling components and frameworks with UML. *Communications of the ACM*, 43:31–39, 2000.
- [cve08] CVE Statistics, 2008. <http://web.nvd.nist.gov/view/vuln/statistics> Last date accessed 2008-10-07.

- [DD07] David Dittrich and Sven Dietrich. Command and control structures in malware. *LOGIN*, 32, 2007.
- [Den68] Peter J. Denning. Thrashing: Its causes and prevention. In *Fall Joint Computer Conference*, pages 915–922, 1968.
- [Eil05] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley, 2005.
- [Erd04] Gergely Erdélyi. Hide’n’sseek? anatomy of stealth malware. *Virus Bulletin Conference*, 2004.
- [ext01] VMware’s installation magic, 2001. <http://www.extremetech.com/article2/0,2845,1156611,00.asp> Last date accessed 2008-08-28.
- [Fow03] Marin Fowler. *UML Distilled*. Addison-Wesley, 3 edition, 2003.
- [GAWF07] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *HotOS 2007*, 2007.
- [Hau07] Lars Haukli. Analysing malicious code. Master’s thesis, The Norwegian University of Science and Technology, 2007.
- [HC03] Neal Hindocha and Eric Chien. Malicious threats and vulnerabilities in instant messaging. White paper, Symantec Security Response, 2003. <http://www.internetsymantec.com/avcenter/reference/malicious.threats.instant.messaging.pdf> Last date accessed 2008-09-02.
- [Hei04] Jay G. Heiser. Understanding today’s malware. Technical report, TruSecure Ltd, 2004.
- [KBS⁺07] Kris-Mikael Krister, Egil Trygve Baadshaug, Daniele Spampinato, Eilev Hagen, Ketil Velle, and Eirik Reksten. SeaMonster - a security modeling software, 2007. http://folk.ntnu.no/krismika/seamonster/seamonster_final_report-web.pdf Last date accessed 2008-09-02.
- [KCV05] S. Katzenbeisser, C. schallhart, and H. Veith. Malware engineering. Institut für Informatik, Technische Universität München, 2005.
- [Lea05] Neal Leavitt. Instant messaging: a new target for hackers, 2005.
- [LL08] Thomas Langerud and Jøran Vagnby Lillesand. Powerscan: A framework for dynamic analysis and anti-virus based identification of malware. Master’s thesis, The Norwegian University of Science and Technology, 2008.

- [McG06] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley, 2006.
- [PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17:412–421, 1974.
- [Pla07] Johannes Plachy. The portable executable file format, 2007. <http://http://www.csn.ul.ie/~caolan/publink/winresdump/winresdump/doc/pefile.html>, Last date accessed 2008-10-07.
- [Pre00] Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl, 2000.
- [QS00] Danny Quist and Val Smith. Detecting the presence of virtual machines, 2000. <http://www.offensivecomputing.net/files/active/0/vm.pdf> Last date accessed 2008-09-01.
- [Rub06] Paul Rubens. Server virtualization goes prime time, 2006. <http://www.serverwatch.com/tutorials/article.php/3637631> Last date accessed 2008-12-09.
- [Rut04] Joanna Rutkowska. Red pill...or how to detect VMM using (almost) one CPU instruction, 11 2004. <http://invisiblethings.org/papers/redpill.html>, Last date accessed 2008-09-01.
- [Sca07] Joel Scambray. *Hacking Exposed Windows: Microsoft Windows Security Secrets and Solutions*. McGraw-Hill Osborne Media, 2007.
- [Sch99] Gerald Scheidl. Virus naming convention 1999 (VNC99), 1999. <http://members.chello.at/erikajo/vnc99b2.txt>, Last date accessed 2008-08-28.
- [Sch02] Markus Schmall. Heuristic techniques in AV solutions: An overview, 2002. <http://www.mschmall.de/heuristic.pdf> Last date accessed 2008-08-28.
- [SF01] Péter Ször and Peter Ferrie. Hunting for metamorphic. White paper, Symantec Security Response, 2001. <http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf> Last date accessed 2008-08-28.
- [SPF08] Randal L. Schwartz, Tom Phoenix, and Brian D. Foy. *Learning Perl*. O'Reilly Media, 5 edition, 2008.
- [SS72] Michael Schroeder and Jerome Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3), 1972.

- [sys06] Microsoft acquires wininternals, 2006. <http://www.microsoft.com/systemcenter/wininternals.aspx> Last date accessed 2008-08-28.
- [Tan01] Andrew Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2001.
- [vir91] A new virus naming convention, 1991. <http://vx.netlux.org/lib/asb01.html> Last date accessed 2008-08-28.
- [vir05] eTrust antivirus malware naming policy, 2005. <http://ca.com/us/securityadvisor/documents/collateral.aspx?cid=71549> Last date accessed 2008-08-28.
- [vmw06] Programming API programming guide, 2006. http://www.vmware.com/pdf/Prog_API_Prog_Guide.pdf, Last date accessed 2008-09-03.
- [vmw07] Understanding full virtualization, paravirtualization, and hardware assist, 2007. http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf Last date accessed 2008-08-28.
- [vWM05] Kenneth R. van Wyk and Gary McGraw. Bridging the gap between software development and information security. *IEEE Security and Privacy*, 3(5):75–79, 2005.
- [WFLY04] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. <http://eprint.iacr.org/>.
- [wha08] What is VIX and why does it matter?, 2008. <http://blogs.vmware.com/vix/2008/07/what-is-vix-and.html>, Last date accessed 2008-09-03.
- [wra] Wrapping your head around virtualization. [http://www.pps-systeme.de/itmanagement.nsf/D9CD1F1C6AA58E46C12573FA004B7A4F/\\$File/wrapping_head_around_virtualization.pdf](http://www.pps-systeme.de/itmanagement.nsf/D9CD1F1C6AA58E46C12573FA004B7A4F/$File/wrapping_head_around_virtualization.pdf) Last date accessed 2008-09-01.

Appendix A

User manual

This chapter covers a step by step guide to set up and use the implemented system described in this report. The detail are in such a level that this text can be considered a complete review in getting the system operational and running. Code snippets are supplied where appropriate in `typewriter` text. This chapter assumes a system specification as follows. Note that different system specifications might work just as fine, but the described specifications reflects the test system.

- A 32bit single-processor system installed with Ubuntu Linux 8.10 fully patched prior to date 2008-11-06.
- `VMware Server 2.0.0` to be installed on the host operating system.
- Optionally a separate machine where the `vmcom` program is located.
- Minimum 1GB of memory on the machine(s).
- `Perl v5.10.0` to be installed on the host operating system and all systems using `vmcom`.

A.1 Setting up and configuring the system

1. As the `vmcom`-system is programmed in Perl, this software has to be installed. To do so, run the following command.

```
$ sudo aptitude update
$ sudo aptitude install perl
$ sudo perl -MCPAN -e shell
$ cpan> install forks
$ cpan> install forks::shared
$ cpan> install Perl::Unsafe::Signals
```

2. The same applies for the virtual machine server. **VMware Server** can be downloaded from <http://www.vmware.com/freedownload/login.php?product=server20>. Run the installation as explained in the documentation. Remember to enable networking. A NAT-interface is recommended to make things easy.
3. Next up, the Perl bindings for the VIX-API must be configured. Enter its directory located in **VMWARE-SERVER-PATH/vmware-vix/** and run the following commands.

```
$ tar zxvf vix-perl.tar.gz $ cd vix-perl
$ perl Makefile.PL
$ make
$ sudo make install
$ echo "/usr/lib/vmware/vmware-vix/api/vix-perl" >> \
/etc/ld.so.conf.d/libc.conf
$ sudo ldconfig
$ sudo ln -s /usr/lib/libcrypto.so.0.9.8 \
/usr/bin/libcrypto.so.0.9.7
$ sudo ln -s /usr/lib/libssl.so.0.9.8 \
/usr/bin/libssl.so.0.9.7
```

If this machine is located remotely from the virtual machine, copy this directory to the remote machine. There is also need for some additional steps. Download the VIX-API from <http://www.vmware.com/download/sdk/vmauto.html> and store it in an appropriate location. Follow the next commands.

```
$ tar zxf VMware-vix-1.1.5-109488.i386.tar.gz
$ cd vmware-vix-distrib
$ sudo perl vmware-install.pl
```

4. Copy the **vmcom** program (**vmcom.pl**) from its web page <http://www.sourceforge.net/projects/vmcom> and store it in an appropriate location.
5. Launch the **VMware Web Access** (a graphical user interface shipped with **VMware Server**) by issuing the command as follows.

```
$ vmware
```

Add a new virtual machine and install a fresh copy on Windows XP onto it (study the documentation shipped with **VMware Server**). Remember

to configure it properly as this would be the basic image the rest of the virtual machines are built from. The program **VMware tools** has to be installed for **vmcom** to work. See the documentation for **VMware Server** on how to installing **VMware tools**.

6. Disable the floppy drive for the image¹ The reason for this is that only one of the **VMware** images can use the floppy drive at once.
7. Copy the installation image and *add it to the inventory* through the **VMware Server** application. See the VMware documentation for more information.

```
$ cp -r /path/to/virtual_machines/clean_image \
/path/to/virtual_machines/first_operational_virtual_machine
```

8. Now install a selected antivirus product, and update **vmcom.pl** with appropriate values. The template shown in A.1 must be used and added in the file to enable the virtual machine.

```
1  AvProduct->new(name      => 'Product name',
2  scan_command   => "/C \"c:\\scan.bat\"",
3  update_command => "/C \"c:\\update.bat",
4  uri            => 'https://127.0.0.1:8333/sdk',
5  username       => 'vmware_admin_user',
6  password       => 'vmware_admin_password',
7  imagepath      => '[standard] relative_path_to/image.vmx',),
```

Listing A.1: Perl code template for adding and enabling a virtual machine.

9. Repeat the last step for all antivirus products.

A.2 Available command line switches

The behaviour of the program is controled via *command line switches*. This is string arguments supplied to the program at launch time, and gives the possibility of automating different tasks from the same program. By running the program with an empty set of switches, the complete set is shown with a brief description. Most of the switches can be trimmed. For example, the switch to display configured antivirus products, **listproducts**, is equivalent with **list**, **li** and so on. The **verbose** switch can be used for a more comprehensive output if needed. **vmcom** outputs to **stdout**, with content depending on which command line switch supplemented.

¹Disabling the floppy drive may be important as the virtual machine might suddenly stop working otherwise. At least that was the experience from this project.

A.2.1 Request a file scan

Initialise the file scanning process by passing the relevant command line switch supplemented with the file to scan. Wait until the results are shown. A more comprehensive output is used when the **verbose** switch is enabled.

```
$ ./vmcom.pl --scan /path/to/file --verbose
```

A normal file scan will output a list of comma separated values as shown below. The first field is the product name, the second is information of any threat found. The third is an info field, while the last is a boolean field if any error was found. In the example below, the antivirus test file “Eicar.com” was used as a sample file. The Eicar-file is a file recognised by practically all antivirus products and can be used to test detection without working with dangerous files.

```
$ ./vmcom.pl --scan /tmp/eicar.com

PRODUCT,THREAT FOUND,INFO,ERROR
Avira AntiVir,-1,Error,1
AVG Free,EICAR_Test,0,0
Panda Engine 1.4.3,EICAR-AV-TEST-FILE,0,0
ClamWin,Eicar-Test-Signature FOUND,0,0
BitDefender Free,-1,Error,1
Kaspersky Anti-Virus,EICAR-Test-File,0,0
F-Secure AntiVirus,EICAR-Test-File,0,0
ESET NOD32,Eicar test file,0,0
Trend Micro,Eicar_test_file,0,0
Avast Professional,EICAR Test-NOT virus!!,0,0
McAfee,EICAR test file NOT a virus.,0,0
```

A.2.2 Performing operations on a single product

To enable operations on one virtual machine only, supplement the command line switch to the operation. Consider the following file scan example where only the first listed antivirus product is used to scan the file.

```
$ ./vmcom.pl --scan /path/to/file --only 1
```

By using the verbosity-switch, more output will be listed. See below for an example output when running a normal file scan with only one antivirus product. The bracket with a number on each line signifies the amount of seconds from the operation start.

```
$ ./vmcom.pl --scan /tmp/eicar.com --verbose --only 9
```

```
[0] - Initiating file scan on /tmp/eicar.com
```

```

[0] - ESET NOD32 - Connecting to virtual machine.
[1] - ESET NOD32 - Opening virtual machine handle.
[2] - ESET NOD32 - Powering on.
[57] - ESET NOD32 - VM not ready. Waiting...
[61] - ESET NOD32 - Authenticating user to VM OS
[63] - ESET NOD32 - Copying file from host to guest.
[63] - ESET NOD32 - Running program...
[68] - ESET NOD32 - Copying file from host to guest.
[69] - ESET NOD32 - Command finished with results: Eicar test file
[69] - ESET NOD32 - Suspending VM...
PRODUCT,THREAT FOUND,INFO,ERROR
ESET NOD32,Eicar test file,0,0

```

A.2.3 List available products

The indexes used together with the “only” switch shown above, is the product index gained from the list product-switch. See below for the current test system used for this project.

```

$ ./vmcom.pl --listproducts
Product index 1 => F-Secure AntiVirus
Product index 2 => AVG Free
Product index 3 => BitDefender Free
Product index 4 => Avast Professional
Product index 5 => Kaspersky Anti-Virus
Product index 6 => Panda Engine 1.4.3
Product index 7 => Trend Micro
Product index 8 => McAfee
Product index 9 => ESET NOD32
Product index 10 => Avira AntiVir
Product index 11 => ClamWin

```

A.2.4 Request virus definition updates

Initialise the update process by passing the relevant command line switch. Wait until the results are shown.

```
$ ./vmcom.pl --update --verbose
```

A.2.5 Handle virtual machine states

Initialise saving of current system states by passing the relevant command line switch. Wait until the results are shown.

```
$ ./vmcom.pl --takesnapshot --verbose
```

To restore a previously saved system state, simply supply the relevant command line switch.

```
$ ./vmcom.pl --revert --verbose
```

To manually suspend virtual machines, pass the relevant command line switch. This will not be necessary in most occasions, as `vmcom` will try to suspend virtual machines whenever appropriate or required to do so.

```
$ ./vmcom.pl --suspend --verbose
```

To manually power off virtual machines, pass the relevant command line switch. A power off is similar to a hard system reboot, and should be avoided whenever possible to avoid damaging the virtual machine file system.

```
$ ./vmcom.pl --poweroff --verbose
```

A.2.6 Avoid suspending the virtual machine(s)

To avoid suspending the virtual machines after its execution, supply the relevant command line switch.

```
$ ./vmcom.pl --update --nosuspend --verbose
```

```
$ ./vmcom.pl --scan /tmp/eicar.com --nosuspend --verbose
```


Appendix B

vmcom in detail

This appendix contains the complete source code of the second part of the system, merely the `vmcom` program. It is listed entirely in Section B.1. Details regarding on demand arguments and scripts are covered in Section B.2, and an activity diagram showing program flow can be found in Section B.3.

B.1 Complete source code

See Listing B.1 below for the complete Perl source code for `vmcom`. The program can also be downloaded as a regular file from <http://www.sourceforge.net/projects/vmcom>.

```
1  #!/usr/bin/perl
2
3  # Run in strict mode
4  use strict;
5
6  # Give warnings
7  use warnings;
8
9  # We need case-switch functionality in the parser
10 use Switch;
11
12 # We also need command line switch parsing
13 use Getopt::Long;
14
15 # Output can be colored
16 use Term::ANSIColor;
17
18 # Thread module that do not require recompilation of Perl
19 use forks;
20 use forks::shared;
21
22 # For timeout to work, the unsafe signals must be used
23 use Perl::Unsafe::Signals;
24
```

```

25 # VIX
26 use VMware::Vix::Simple;
27 use VMware::Vix::API::Constants;
28
29 # Needed to at least have some structure in the product
    configuration
30 use Class::Struct;
31
32 # Constants
33 use constant SCAN_FILE => 1;
34 use constant UPDATE_DEFINITIONS => 2;
35 use constant REVERT_SNAPSHOTS => 3;
36 use constant VM_SUSPEND => 4;
37 use constant REVERT_SNAPSHOT => 5;
38 use constant POWER_OFF => 6;
39 use constant TAKE_SNAPSHOT => 7;
40
41 use constant SCAN_POSITIVE => 20;
42 use constant SCAN_NEGATIVE => 21;
43 use constant SCAN_ERROR => 22;
44 use constant VM_SUSPENDED => 23;
45 use constant VM_POWEREDOFF => 24;
46 use constant VM_SNAPSHOTOK => 25;
47
48 # #####
49 #
50 # CONFIGURATION PART
51 #
52
53 # Defining the class structure consiting of strings for each
    antivirus
54 # product (AvProduct)
55 struct AvProduct =>
56 {
57     name => '$',
58     scan_command => '$',
59     update_command => '$',
60     uri => '$',
61     username => '$',
62     password => '$',
63     imagepath => '$',
64
65 };
66
67
68 # Initialise the set of antivirus products on this system and store
    them
69 # all in an array.
70 my @av_products = (
71     AvProduct->new(name => 'F-Secure AntiVirus',
72         scan_command => "/C \"c:\\program files\\f-secure\\
            anti-virus\\fsav.exe\" c:\\scanme > c:\\results.txt",
73         update_command => undef, # Deprecated, use automatic
            updates instead
74         uri => 'https://127.0.0.1:8333/sdk',
75         username => 'root',
76         password => 'SCRAMBLED',

```

```

77         imagepath      => '[standard] WinXP_2/WinXP_2.vmx',
78     ),
79
80     AvProduct->new(name      => 'AVG Free',
81         scan_command    => "/C \"c:\\program files\\AVG\\AVG8\\
            avgscanx.exe\" /SCAN=c:\\scanme > c:\\results.txt",
82         update_command  => "/C \"c:\\program files\\AVG\\AVG8\\
            avgupd.exe /SCHED=4\"", # Works, but does not give
            any textual results
83         uri             => 'https://127.0.0.1:8333/sdk',
84         username        => 'root',
85         password        => 'SCRAMBLED',
86         imagepath      => '[standard] WinXP_3/WinXP_3.vmx',
87     ),
88
89     AvProduct->new(name      => 'BitDefender Free',
90         scan_command    => "/C \"c:\\program files\\Common Files
            \\Softwin\\Bitdefender Scan Server\\bdc.exe\" c:\\
            scanme /f > c:\\results.txt",
91         update_command  => "/C \"c:\\program files\\Common Files
            \\Softwin\\Bitdefender Scan Server\\bdc.exe\" /update
            > c:\\results.txt", # Works
92         uri             => 'https://127.0.0.1:8333/sdk',
93         username        => 'root',
94         password        => 'SCRAMBLED',
95         imagepath      => '[standard] WinXP_4/WinXP_4.vmx',
96     ),
97
98     AvProduct->new(name      => 'Avast Professional',
99         # See ashcmd help page for details regarding the
            arguments
100         scan_command    => "/C \"c:\\program files\\Alwil
            Software\\Avast4\\ashcmd.exe\" /_ /a /s /t =
            JZIMXRSTGCBWOEQHFVKPY7D6UAN1 c:\\scanme > c:\\results
            .txt",
101         update_command  => "/C \"c:\\program files\\Alwil
            Software\\Avast4\\ashupd.exe\" program /silent /
            autoreboot > c:\\results.txt", # Works
102         uri             => 'https://127.0.0.1:8333/sdk',
103         username        => 'root',
104         password        => 'SCRAMBLED',
105         imagepath      => '[standard] WinXP_5/WinXP_5.vmx',
106     ),
107
108     AvProduct->new(name      => 'Kaspersky Anti-Virus',
109         scan_command    => "/C \"c:\\program files\\Kaspersky
            Lab\\Kaspersky Anti-Virus 7.0\\avp.com\" SCAN c:\\
            scanme /i0 > c:\\results.txt",
110         update_command  => undef, # Script do not download files
            to the correct location
111         uri             => 'https://127.0.0.1:8333/sdk',
112         username        => 'root',
113         password        => 'SCRAMBLED',
114         imagepath      => '[standard] WinXP_6/WinXP_6.vmx',
115     ),
116
117     AvProduct->new(name      => 'Panda Engine 1.4.3',

```

```

118     # No boot sectors, heuristic on,
119     scan_command => "/C \"c:\\program files\\Panda
        Software\\cmd_line\\pavcl.exe\" -auto -nob -cmp -nos
        -heu -rpt:c:\\results.txt -aex c:\\scanme",
120     update_command => undef, # Dateset do not work (needed
        for script to work)
121     uri => 'https://127.0.0.1:8333/sdk',
122     username => 'root',
123     password => 'SCRAMBLED',
124     imagepath => '[standard] WinXP_7/WinXP_7.vmx',
125 ),
126
127 AvProduct->new(name => 'Trend Micro',
128     scan_command => "/C \"c:\\program files\\Trend Micro
        \\Internet Security\\vscantm.bin\" /nm /nb /nc\\[2\\] c
        :\\scanme > c:\\results.txt",
129     update_command => undef, # Not possible ?
130     uri => 'https://127.0.0.1:8333/sdk',
131     username => 'root',
132     password => 'SCRAMBLED',
133     imagepath => '[standard] WinXP_8/WinXP_8.vmx',
134 ),
135
136 AvProduct->new(name => 'McAfee',
137     # No memory scan. Report to results.txt
138     scan_command => "/C \"c:\\mcafee_cmd_line\\scan.exe /
        NOMEM /REPORT C:\\results.txt /RPTALL c:\\scanme\"",
139     update_command => undef, # Not possible ?
140     uri => 'https://127.0.0.1:8333/sdk',
141     username => 'root',
142     password => 'SCRAMBLED',
143     imagepath => '[standard] WinXP_9/WinXP_9.vmx',
144 ),
145
146 AvProduct->new(name => 'ESET NOD32',
147     # Scan command unavailable in a one-liner. Using script
        located on the virtual machine
148     scan_command => "/C \"c:\\scan.bat\"",
149     update_command => undef, # Must use scheduler or
        automatic updates
150     uri => 'https://127.0.0.1:8333/sdk',
151     username => 'root',
152     password => 'SCRAMBLED',
153     imagepath => '[standard] WinXP_10/WinXP_10.vmx',
154 ),
155
156 AvProduct->new(name => 'Avira AntiVir',
157     scan_command => "/C \"c:\\program files\\Avira\\
        AntiVir PersonalEdition Classic\\avcls.exe\" -noboot
        -nombr -rl -ro -rfc:\\results.txt c:\\scanme < c:\\
        avira_cmd\\cr.txt",
158     update_command => "/C \"c:\\program files\\Avira\\
        AntiVir PersonalEdition Classic\\preupd.exe\", # Works
        , but have to pause before suspend (program
        immediately exits)
159     uri => 'https://127.0.0.1:8333/sdk',
160     username => 'root',

```

```

161         password      => 'SCRAMBLED',
162         imagepath      => '[standard] WinXP_11/WinXP_11.vmx',
163     ),
164
165     AvProduct->new(name      => 'ClamWin',
166         # Scan command unavailable in a one-liner. Using script
167         # located on the virtual machine
168         scan_command      => "/C \"c:\\scan.bat\"",
169         # Update command unavailable in a one-liner. Using script
170         # located on the virtual machine
171         update_command     => "/C \"c:\\update.bat", # Works
172         uri                => 'https://127.0.0.1:8333/sdk',
173         username           => 'root',
174         password           => 'SCRAMBLED',
175         imagepath          => '[standard] WinXP_12/WinXP_12.vmx',
176     ),
177 );
178
179 share ( @av_products );
180
181 # Set the timeout for each virtual machine. When the operations on
182 # a vm
183 # combined has used more time than the timeout, the program will
184 # receive
185 # an alarm.
186
187 # Set the timeout for each virtual machine operation. After a
188 # timeout,
189 # the current operation will stop and cleanup will be done
190 # according to
191 # what kind of operation timed out.
192 #
193 # If the operation was a vm_suspend, do a rollback
194 # else do a vm_suspend
195 use constant TIMEOUT => 220;
196 use constant CONCURRENT_RUNNING_VMS => 1;
197
198 #
199 # END CONFIGURATION PART
200 #
201 #####
202
203 #####
204 #
205 # MAIN PROGRAM
206 #
207
208 # Display a help screen if:
209 # a) user have supplemented the argument --help
210 # b) no arguments are specified
211
212 print_help() unless ($ARGV[0]);
213
214 # This is the Getopts switch hash, configuring the legal command
215 # line
216 # switches. The switches "only" and "scan" requires an argument

```

```

211     each,
212 # respectively a string and an integer.
213 my %switches = ();
214 GetOptions(
215     "only=i"=>\$switches{only},
216     "poweroff!"=>\$switches{poweroff},
217     "scan=s"=>\$switches{scan},
218     "update!"=>\$switches{update},
219     "revert!"=>\$switches{revert},
220     "suspend!"=>\$switches{suspend},
221     "takesnapshot!"=>\$switches{takesnapshot},
222     "verbose!"=>\$switches{verbose},
223     "listproducts!"=>\$switches{listproducts},
224     "help!"=>\$switches{help},
225     "nosuspend!"=>\$switches{nosuspend},
226 );
227 print_help() if \$switches{help};
228
229 # If there are any arguments left after getopt has parsed the
230 # arguments, some of them was not accepted by getopt. Inform the
231 user
232 # accordingly.
233 foreach (@ARGV) {
234     if (defined(\$switches{$_})) {
235         print "$_ = \$switches{$_}\n";
236     }
237     else {
238         print "$_ ";
239     }
240 }
241
242 my $type = undef;
243 # Set constant values for code readability
244 if (\$switches{scan}) { $type = SCAN_FILE }
245 elsif (\$switches{update}) { $type = UPDATE_DEFINITIONS }
246 elsif (\$switches{suspend}) { $type = VM_SUSPEND }
247 elsif (\$switches{revert}) { $type = REVERT_SNAPSHOT }
248 elsif (\$switches{poweroff}) { $type = POWER_OFF }
249 elsif (\$switches{takesnapshot}) { $type = TAKE_SNAPSHOT }
250 elsif (!\$switches{listproducts}) { die "Supplement either --scan,
    --update, --suspend, --revert, --poweroff, --takesnapshot or --
    list\n"; }
251
252 # Global shared hash containing operation results
253 my %results = ();
254 # Global shared scalar containing current running virtual machines
255 from
256 # this vmcom-session
257 my $running_vms = 0;
258 # Global shared scalar containing current completed virtual
259 machines
260 # from this vmcom-session
261 my $completed_vms = 0;
262 share( %results );
263 share( $running_vms );
264 share( $completed_vms );

```

```

262
263 # Note the current time
264 my $start = time;
265 my $subject_file;
266
267 # Seems like this is a file scan job
268 if ($type and $type == SCAN_FILE) {
269     # Get the file path to scan and ensure it is readable
270     $subject_file = $switches{scan};
271     terminal_output(undef, "Initiating file scan on $subject_file");
272     unless (-r $subject_file and -e $subject_file) {
273         # Kill if the file cannot be read or the file does not exist
274         die "Cannot read file $subject_file\n";
275     }
276 }
277
278
279 my $av_command = undef;
280
281 # List product indices to the user on request
282 if ($switches{listproducts}) {
283     for (my $i = 0; $i < scalar(@av_products); $i++) {
284         print "Product index " . ($i+1) . " => " . $av_products[$i]->name . "\n";
285     }
286 }
287
288 # If the --only switch is specified, do the operation on one VM
289 # only
290 elseif ($switches{only}) {
291     # To convert the index to the perl zero-based values, subtract 1
292     my $product_index = $switches{only}-1;
293     # Check if the index is out of bound
294     die "Invalid index $product_index\n" if $product_index < 0 or
295         $product_index >= scalar(@av_products);
296
297     # Get the current command if the relevant switch is set
298     $av_command = $av_products[$product_index]->update_command
299         if $switches{update};
300     $av_command = $av_products[$product_index]->scan_command
301         if $switches{scan};
302
303     run_av_command_and_parse_results(
304         $type, # Type of operation
305         $av_products[$product_index]->name, # Product name
306         parse_s($av_products[$product_index]->uri), # Hostname
307         0, # Port
308         parse_s($av_products[$product_index]->username), # SSH
309             host username
310         parse_s($av_products[$product_index]->password), # SSH
311             host pw
312         $av_command, # Executable command string
313         parse_s($av_products[$product_index]->imagepath), #
314             Path to VM
315         "km", # Guest OS username
316         "secret", # Guest OS pw

```

```

311     $subject_file ,                                # Relevant file (only used in
312         scan)                                     # Guest file scheduled for
313     "c:\\results.txt",                             # Guest file scheduled for
314         parsing
315     "/tmp/results_". $av_products[$product_index]. ".txt"
316 );
317 }
318
319 else {
320     while (scalar(@av_products) > 0) {
321         if ($running_vms < CONCURRENT_RUNNING_VMS) {
322             # Increment the amount of running virtual machines
323             $running_vms++;
324             my $av_product = shift(@av_products);
325             $av_command = $av_product->update_command if $switches{
326                 update};
327             $av_command = $av_product->scan_command if $switches{
328                 scan};
329
330             threads->new(&run_av_command_and_parse_results,
331                 $type,                                # Type of operation
332                 $av_product->name,                      # Product name
333                 parse_s($av_product->uri),              # Hostname
334                 0,                                     # Port
335                 parse_s($av_product->username),         # SSH host username
336                 parse_s($av_product->password),         # SSH host pw
337                 $av_command,                           # Executable command string
338                 parse_s($av_product->imagepath),        # Path to VM
339                 "km",                                  # Guest OS username
340                 "secret",                              # Guest OS pw
341                 $subject_file,                          # Relevant file (only used in
342                     scan)
343                 "c:\\results.txt",                      # Guest file scheduled for
344                     parsing
345                 "/tmp/results_". $av_product->name. ".txt"
346             );
347         }
348         else {
349             print "[THR] - Waiting for thread to finish. Running
350                 $running_vms/" . CONCURRENT_RUNNING_VMS . " Completed
351                 $completed_vms/" . ($completed_vms + scalar(@av_products)) . "
352                 \n";
353             sleep 10;
354         }
355     }
356 }
357
358 # Clean up the program (join running threads)
359 foreach my $thr (threads->list) {
360     # Don't join the main thread or ourselves
361     if ($thr->tid and !threads::equal($thr, threads->self)) {
362         $thr->join;
363     }
364 }
365
366 # Output results CVS formatted

```



```

359 print("PRODUCT,THREAT FOUND,INFO,ERROR\n");
360 for my $av_product (keys %results) {
361     my $value = $results{$av_product};
362     print("$av_product ,");
363
364     switch ($value) {
365         case SCAN_NEGATIVE { print("0,0,0") }
366         case SCAN_ERROR    { print("-1,Error,1") }
367         case VM_SUSPENDED  { print("-1,Suspended,0") }
368         case VM_POWEREDOFF { print("-1,Powered off,0") }
369         case VM_SNAPSHOTOK { print("-1,Snapshot OK,0") }
370         else                { print("$value,0,0") }
371     }
372     print("\n");
373 }
374
375 #####
376 #
377 # SUBROUTINES
378 #
379
380 # Prints a static help page
381 sub print_help {
382     print("\vmcom, using VIX API Version ".VIX_API_VERSION."\n\n");
383     print("--nosuspend\t\tDo not suspend the virtual machine(s)\n\n");
384     print("--poweroff\t\tPower off all virtual machines, or virtual
machine\n\t\twith index N if --only is specified\n\n");
385     print("--only N\t\tMark index N\n\n");
386     print("--revert\t\tRevert all virtual machines to clean state
from already\n\t\tstored snapshot. Revert only virtual
machine with index N\n\t\tif --only is specified\n\n");
387     print("--scan PATH\t\tScan file located at PATH by using all
antivirus products,\n\t\t\tor product with index N if --only
is specified\n\n");
388     print("--suspend\t\tSuspend all virtual machines, or virtual
machine with\n\t\t\tindex N if --only is specified\n\n");
389     print("--takesnapshot\t\tStore snapshots for the logged in state
of virtual machines,\n\t\t\tor virtual machine with index N
if --only is specified\n\t\t\tWARNING: This WILL overwrite
any existing snapshots\n\t\t\tfor the virtual machines!\n\n");
390     print("--update\t\tUpdate all antivirus products, or product
with index N\n\t\t\tif --only is specified\n\n");
391     print("--verbose\t\tIncrease verbosity level\n\n");
392     print("--listproducts\t\tList all products' indices\n\n");
393     print("--help\t\t\tThis help screen\n\n");
394     die "\n";
395 }
396
397 # Helper function returning proper datatypes according to string
data
398 sub parse_s {
399     switch ($_[0]) {
400         case 0 { return 0 }
401         case "undef" { return undef }
402         else { return "$_[0]" }

```

```

403     }
404 }
405
406 # Connects to virtual machine
407 sub connect_to_vm {
408     my($err,
409        $hostHandle,
410        $host_name,
411        $host_port,
412        $host_username,
413        $host_password,
414        $sav_product) = @_;
415     terminal_output($sav_product, "Connecting to virtual machine.");
416
417     ($err, $hostHandle) = HostConnect(VIX_API_VERSION,
418                                       VIX_SERVICEPROVIDER_VMWARE_VI_SERVER,
419                                       $host_name,
420                                       $host_port,
421                                       $host_username,
422                                       $host_password,
423                                       0, # options
424                                       VIX_INVALID_HANDLE); # propertyListHandle
425
426     abort("HostConnect() failed", $err, $sav_product) if $err !=
427         VIX_OK;
428
429     return $hostHandle;
430 }
431
432 # Opens a virtual machine handle
433 # Requires: connection to a virtual machine (connect_to_vm)
434 sub open_vm {
435     my($err, $hostHandle, $svmHandle, $vmx_location, $sav_product) =
436         @_;
437     terminal_output($sav_product, "Opening virtual machine handle.");
438
439     ($err, $svmHandle) = VMOpen($hostHandle,
440                                $vmx_location); # VM.vmx location
441
442     abort("VMOpen() failed", $err, $sav_product) if $err != VIX_OK;
443
444     return ($hostHandle, $svmHandle);
445 }
446
447 # Power on a virtual machine handle
448 # Opens a virtual machine handle
449 # Requires: connection to a virtual machine (connect_to_vm)
450 # an open virtual machine handle (open_vm)
451 sub power_on_vm {
452     my($err, $svmHandle, $sav_product) = @_;
453     terminal_output($sav_product, "Powering on.");
454
455     $err = VMPowerOn($svmHandle,
456                      0, # powerOnOptions
457                      VIX_INVALID_HANDLE); # propertyListHandle
458
459     abort("VMPowerOn() failed", $err, $sav_product) if $err != VIX_OK

```

```

458     ;
459     $results{$sav_product} = VM_POWEREDOFF;
460 }
461 # Wait until vmware tools is loaded in the virtual machine
462 # Requires: connection to a virtual machine (connect_to_vm)
463 #     an open virtual machine handle (open_vm)
464 #     a powered on virtual machine (power_on_vm)
465 sub wait_for_vm {
466     my($err, $vmHandle, $sav_product) = @_;
467     terminal_output($sav_product, "VM not ready. Waiting...");
468
469     $err = VMWaitForToolsInGuest($vmHandle,
470     TIMEOUT); # timeoutInSeconds
471
472     # vm suspend
473     abort("VMWaitForToolsInGuest() failed", $err, $sav_product) if
474         $err != VIX_OK;
475 }
476
477 # Log into virtual machine guest operating system
478 # Requires: connection to a virtual machine (connect_to_vm)
479 #     an open virtual machine handle (open_vm)
480 #     a powered on virtual machine (power_on_vm)
481 #     vmware tools loaded (log_in_vm)
482 sub log_in_vm {
483     my($err, $vmHandle, $guest_os_username, $guest_os_password,
484     $sav_product) = @_;
485     terminal_output($sav_product, "Authenticating user to VM OS");
486
487     $err = VMLoginInGuest($vmHandle,
488     $guest_os_username, # userName
489     $guest_os_password, # password
490     0); # options
491     # vm suspend
492     abort("VMLoginInGuest() failed", $err, $sav_product) if $err !=
493         VIX_OK;
494 }
495
496 # Copy a file from host operating system to guest operating system
497 # Requires: connection to a virtual machine (connect_to_vm)
498 #     an open virtual machine handle (open_vm)
499 #     a powered on virtual machine (power_on_vm)
500 #     vmware tools loaded (log_in_vm)
501 #     a user logged into the virtual machine guest OS (log_in_vm)
502 sub copy_file_to_guest {
503     my($err, $vmHandle, $subject_file_location, $sav_product) = @_;
504     terminal_output($sav_product, "Copying file from host to guest.");
505     ;
506
507     $err = VMCopyFileFromHostToGuest($vmHandle,
508     $subject_file_location, # src name
509     "c:\\scanme", # dest name
510     0, # options
511     VIX_INVALID_HANDLE); # propertyListHandle
512     # vm suspend

```

```

510     abort("VMCopyFileFromHostToGuest() failed", $err, $sav_product)
511     if $err != VIX_OK;
512 }
513 # Delete the result file in the guest
514 # Requires: connection to a virtual machine (connect_to_vm)
515 #   an open virtual machine handle (open_vm)
516 #   a powered on virtual machine (power_on_vm)
517 #   vmware tools loaded (log_in_vm)
518 #   a user logged into the virtual machine guest OS (log_in_vm)
519 sub delete_result_file_in_guest {
520     my($err, $vmHandle, $subject_file_location, $sav_product) = @_;
521     $err = VMDeleteFileInGuest($vmHandle, $subject_file_location);
522     terminal_output($sav_product, "Deleting result file in guest.");
523 }
524
525 # Run an arbitrary program on the guest operating system with the
526 #   logged
527 #   in users privileges
528 # Requires: connection to a virtual machine (connect_to_vm)
529 #   an open virtual machine handle (open_vm)
530 #   a powered on virtual machine (power_on_vm)
531 #   vmware tools loaded (log_in_vm)
532 #   a user logged into the virtual machine guest OS (log_in_vm)
533 sub run_vm_program {
534     my($err, $vmHandle, $type, $sav_command, $sav_product) = @_;
535     terminal_output($sav_product, "Running program...");
536
537     $err = VMRunProgramInGuest($vmHandle,
538         "C:\\windows\\System32\\cmd.exe",
539         $sav_command,
540         0, # options
541         VIX_INVALID_HANDLE);
542     # vm suspend
543     abort("VMRunProgramInGuest() failed", $err, $sav_product) if $err
544         != VIX_OK;
545 }
546
547 # Copy a file from the guest operating system to the host
548 # Requires: connection to a virtual machine (connect_to_vm)
549 #   an open virtual machine handle (open_vm)
550 #   a powered on virtual machine (power_on_vm)
551 #   vmware tools loaded (log_in_vm)
552 #   a user logged into the virtual machine guest OS (log_in_vm)
553 sub copy_file_to_host {
554     my($err, $vmHandle, $result_file_source,
555         $result_file_destination,
556         $sav_product) = @_;
557     terminal_output($sav_product, "Copying file from host to guest.");
558     ;
559
560     $err = VMCopyFileFromGuestToHost($vmHandle,
561         $result_file_source, # src name
562         $result_file_destination, # dest name
563         0, # options
564         VIX_INVALID_HANDLE); # propertyListHandle
565     # vm suspend

```

```

562     abort("VMCopyFileFromGuestToHost() failed", $err, $av_product)
563         if $err != VIX_OK;
564 }
565 # Suspend a virtual machine
566 # Requires: connection to a virtual machine (connect_to_vm)
567 #           an open virtual machine handle (open_vm)
568 sub suspend_vm {
569     my($err, $vmHandle, $snapshotHandle, $av_product) = @_;
570     terminal_output($av_product, "Suspending VM...");
571
572     $err = VMSuspend($vmHandle,
573         0); # must be 0
574     if ($err != VIX_OK) {
575         $results{$av_product} = SCAN_ERROR;
576         terminal_output(undef, "VMSuspend() failed, already suspended?
577             ".$err) if $err != VIX_OK;
578         #revert_snapshot($err, $vmHandle, $snapshotHandle);
579     }
580     else {
581         $results{$av_product} = VM_SUSPENDED if $type == VM_SUSPEND;
582     }
583 }
584 # Store the system state of a virtual machine
585 # Requires: connection to a virtual machine (connect_to_vm)
586 #           an open virtual machine handle (open_vm)
587 sub take_snapshot {
588     my($err, $vmHandle, $snapshotHandle, $options, $av_product) =
589         @_;
590     terminal_output($av_product, "Snapshotting...");
591
592     ($err, $snapshotHandle) = VMCreateSnapshot($vmHandle,
593         undef, # name
594         undef, # description
595         $options, # options
596         VIX_INVALID_HANDLE);
597
598     abort("VMCreateSnapshot() failed", $err, $av_product) if $err !=
599         VIX_OK;
600     $results{$av_product} = VM_SNAPSHOTOK;
601 }
602 # Revert to a previously taken system state of a virtual machine
603 # Requires: connection to a virtual machine (connect_to_vm)
604 #           an open virtual machine handle (open_vm)
605 #           previously stored snapshot (take_snapshot)
606 sub revert_snapshot {
607     my($err, $vmHandle, $snapshotHandle, $av_product) = @_;
608     terminal_output($av_product, "Reverting snapshot...");
609
610     ($err, $snapshotHandle) = VMGetRootSnapshot($vmHandle,
611         0); # index
612     abort("VMGetRootSnapshot() failed", $err, $av_product) if $err
613         != VIX_OK;
614
615     $err = VMRevertToSnapshot($vmHandle,

```

```

614     $snapshotHandle ,
615     0,                # options
616     VIX_INVALID_HANDLE); # property handle
617     abort("VMRevertToSnapshot() failed", $err, $av_product) if $err
        != VIX_OK;
618     $results{$av_product} = VM_SNAPSHOTOK;
619 }
620
621 # Swedish button on the virtual machine
622 # Requires: connection to a virtual machine (connect_to_vm)
623 # an open virtual machine handle (open_vm)
624 sub power_off {
625     my($err, $vmHandle, $av_product) = @_;
626     terminal_output($av_product, "Powering off VM.");
627     $err = VMPowerOff($vmHandle,
628         0); # powerOnOptions
629     abort("VMPowerOff() failed", $err, $av_product) if $err !=
        VIX_OK;
630
631 }
632
633 # Note the error and end the current running of the virtual machine
634
635 sub abort {
636     my($text, $err, $av_product) = @_;
637     $results{$av_product} = SCAN_ERROR;
638     terminal_output($av_product, "Error received, $text, $err ".
        GetErrorText($err));
639     die $text, $err, GetErrorText($err), "\n";
640 }
641
642 # Main subroutine for scanning files and updating antivirus
        products on demand
643 sub run_av_command_and_parse_results {
644     my($type,
645         $av_product,
646         $host_name,
647         $host_port,
648         $host_username,
649         $host_password,
650         $av_command,
651         $vmx_location,
652         $guest_os_username,
653         $guest_os_password,
654         $subject_file_location,
655         $result_file_source,
656         $result_file_destination) = @_;
657
658
659     # Initiate error and handles
660     my $err = VIX_OK;
661     my $hostHandle = VIX_INVALID_HANDLE;
662     my $vmHandle = VIX_INVALID_HANDLE;
663     my $snapshotHandle = VIX_INVALID_HANDLE;
664     my $clock;
665

```

```

666
667 # Set up the timeout system
668 local $SIG{ALRM} = sub { print "[THR] - Timeout!\n"; $results{
        $sav_product } = SCAN_ERROR; die "timeout"; };
669 # Start the countdown
670 alarm(TIMEOUT);
671 UNSAFE_SIGNALS {
672     eval {
673         print "\n" if $switches{verbose};
674
675         if ($type == SCAN_FILE or $type == UPDATE_DEFINITIONS) {
676             # Die if the command content is empty
677             abort("No command content", $err, $sav_product) unless
                $sav_command;
678         }
679
680         $hostHandle = connect_to_vm( $err,
681             $hostHandle,
682             $host_name,
683             $host_port,
684             $host_username,
685             $host_password,
686             $sav_product);
687         ($hostHandle, $vmHandle) = open_vm( $err,
688             $hostHandle,
689             $vmHandle,
690             $vmx_location,
691             $sav_product);
692
693         # No need to perform more in this eval block
694         die "" if ( $type == VM_SUSPEND or
695             $type == REVERT_SNAPSHOT);
696
697         power_on_vm($err, $vmHandle, $sav_product);
698         if ($type == POWER_OFF) {
699             # No need to perform more in this eval block
700             die "";
701         }
702         wait_for_vm($err, $vmHandle, $sav_product);
703         log_in_vm( $err,
704             $vmHandle,
705             $guest_os_username,
706             $guest_os_password,
707             $sav_product);
708
709         if ($type == TAKE_SNAPSHOT) {
710             take_snapshot($err, $vmHandle, $snapshotHandle, 0,
                $sav_product);
711             # No need to perform more in this eval block
712             die "";
713         }
714
715         copy_file_to_guest( $err,
716             $vmHandle,
717             $subject_file_location,
718             $sav_product) if ($type == SCAN_FILE);
719

```

```

720     run_vm_program($err, $vmHandle, $type, $av_command,
721                   $av_product);
722
723     copy_file_to_host($err, $vmHandle, $result_file_source,
724                     $result_file_destination, $av_product);
725
726     delete_result_file_in_guest($err, $vmHandle,
727                                $result_file_source, $av_product);
728
729     parse_results($type, $av_product, $result_file_destination);
730 };
731 };
732 alarm(0);
733
734 # Intentionally set outside timeout functionality. Should NOT be
735 # aborted.
736 revert_snapshot($err,
737                $vmHandle,
738                $snapshotHandle,
739                $av_product) if ($type == REVERT_SNAPSHOT);
740
741 if ($type == POWER_OFF) { power_off($err, $vmHandle, $av_product)
742 }
743 # Do not suspend if the nosuspend-option is given
744 elseif (!$switches{nosuspend}) {
745     local $SIG{ALRM} = sub { print "[THR] - Timeout!\n"; $results
746                               { $av_product } = SCAN_ERROR; die "timeout"; };
747     alarm(TIMEOUT);
748     UNSAFE_SIGNALS {
749         eval {
750             suspend_vm($err,
751                       $vmHandle,
752                       $snapshotHandle,
753                       $av_product) if $vmHandle;
754         };
755     };
756 }
757 alarm(0);
758
759 ReleaseHandle($snapshotHandle);
760 ReleaseHandle($vmHandle);
761 HostDisconnect($hostHandle);
762 # Decrement the amount of running virtual machines
763 $running_vms--;
764 # Increment the amount of completed virtual machines
765 $completed_vms++;
766 }
767
768 # Routine for outputting information if we're in verbose mode
769 sub terminal_output {
770     if ($switches{verbose}) {
771         my($av_product,
772            $text,
773            $no_newline) = @_;
774         my $now = time - $start;

```



```

774     if ($av_product) {
775         print("[ $now] - $av_product - $text");
776     }
777     else {
778         print("[ $now] - $text");
779     }
780     print "\n" unless ($no_newline);
781 }
782 }
783 #
784 # END
785 #
786 #####
787
788 #####
789 #
790 # PARSER PART
791 #
792
793 sub parse_results {
794     my $results = undef;
795     my($type,
796        $av_product,
797        $result_file_destination) = @_ ;
798
799     open(RESULTS, $result_file_destination) || die "Could not open
800        file!";
801     my @data = <RESULTS>;
802     close(RESULTS);
803
804     switch ($av_product) {
805
806     case "F-Secure AntiVirus" {
807         foreach my $line (@data) {
808             $results = "clean" if $line =~ "No malware found";
809             if ($line =~ "Infection:") {
810                 my ($text, $data) = split(/Infection: /, $line);
811                 my ($name) = split(/ Action/, $data);
812                 $results = $name;
813             }
814         }
815     case "AVG Free" {
816         foreach my $line (@data) {
817             my ($text, $data) = split(/:/, $line);
818             $results = "clean" if ($text =~ "Found infections" and
819                $data == "0");
820             if ($line =~ "Virus identified") {
821                 my ($text, $data) = split(/Virus identified /, $line);
822                 $results = $data;
823             }
824         }
825     case "BitDefender Free" {
826         if ($type == SCAN_FILE) {
827             foreach my $line (@data) {

```

```

829         my ($text, $data) = split(/:/, $line);
830         $results = "clean" if ($text =~ "Infected files" and
831             $data == "0");
832         if ($line =~ "infected:") {
833             ($text, $data) = split(/infected: /, $line);
834             $results = $data;
835         }
836     }
837
838     elsif ($type == UPDATE_DEFINITIONS) {
839         foreach my $line (@data) {
840             my ($text, $data) = split(/:/, $line);
841             $results = "updated" if ($text =~ "updated");
842         }
843         $results = "update failed" unless $results;
844     }
845 }
846
847 case "Avast Professional" {
848     if ($type == SCAN_FILE) {
849         chomp($data[0]);
850         my $line = $data[0];
851
852         if ($line =~ "OK") {
853             $results = "clean"
854         }
855         else {
856             my ($text, $data) = split(/\t/, $line);
857             $results = $data;
858         }
859     }
860     elsif ($type == UPDATE_DEFINITIONS) {
861         foreach my $line (@data) {
862             $results = "update failed" if ($line =~ "failed");
863         }
864         $results = "updated" unless $results;
865     }
866 }
867
868 case "Kaspersky Anti-Virus" {
869     foreach my $line (@data) {
870         if ($line =~ "detected") {
871             my ($text, $data) = split(/detected/, $line);
872             $results = $data if $text =~ "scanme";
873         }
874     }
875     $results = "clean" unless $results;
876 }
877
878 case "Panda Engine 1.4.3" {
879     foreach my $line (@data) {
880         my ($text, $data) = split(/:/, $line);
881         $results = "clean" if ($text =~ "Number of files infected"
882             and $data == "0");
883         $results = $data if ($text =~ "Found virus");
884     }

```

```

884     }
885
886     case "Trend Micro" {
887         foreach my $line (@data) {
888             $results = "clean" if ($line =~ "No file-type viruses found
889                 .");
890
891             my ($text, $data) = split(/\[/, $line);
892             if ($text =~ "Found Virus") {
893                 chomp($data);
894                 $results = substr($data, 0, -2);
895             }
896         }
897     }
898
899     case "McAfee" {
900         foreach my $line (@data) {
901             if ($line =~ "Found: ") {
902                 my ($text, $data) = split(/Found: /, $line);
903                 $results = $data;
904             }
905             $results = "clean" unless $results;
906         }
907     }
908
909     case "ESET NOD32" {
910         foreach my $line (@data) {
911             if ($line =~ "threat=") {
912                 my @text = split(/\"/, $line);
913                 $results = $text[3];
914             }
915             $results = "clean" unless $results;
916         }
917     }
918
919     case "Avira AntiVir" {
920         foreach my $line (@data) {
921             chomp($line);
922             if ($line =~ m/(A.L.E.R.T.)/) {
923                 my ($text, $data) = split(/ /, $line);
924                 chop($data);
925                 $results = $data;
926                 $results =~ s/\\[\\]//g; #remove leading [ and trailing ]
927             }
928             $results = "clean" unless $results;
929         }
930     }
931
932     case "ClamWin" {
933         my ($text, $data) = split(/:/, $data[0]);
934         chomp($data);
935         if ($data =~ /OK/) {
936             $results = "clean";
937         }
938         else {
939             $results = $data;
940         }
941     }

```

```

940     }
941 }
942
943 if ($results) {
944     chomp($results);
945     $results = ~ s/^\s+//; #remove leading spaces
946     $results = ~ s/\s+$//; #remove trailing spaces
947
948     terminal_output($av_product, "Command finished with results: ",
949                     1);
949     if ($results = ~ "clean") {
950         $results{$av_product} = SCAN_NEGATIVE;
951         print color 'bold green';
952     }
953     elsif ($results = ~ "updated") {
954         $results{$av_product} = $results;
955         print color 'bold green';
956     }
957     else {
958         $results{$av_product} = $results;
959         print color 'bold red';
960     }
961     print("$results\n") if $switches{verbose};
962     print color 'reset';
963 }
964 else {
965     $results{$av_product} = SCAN_ERROR;
966     terminal_output($av_product, "Result parse error");
967 }
968
969 # Now delete the file
970 unlink($result_file_destination) || die("Could not delete file
971     $result_file_destination");
972
973 }
974 #
975 # END PARSER PART
976 #
977 # #####

```

Listing B.1: Complete source code for the vmcom program.

B.2 On demand arguments and scripts

Most of the antivirus product requires special arguments to their on demand processes. This section covers each of the arguments used. All products will scan compressed files and use heuristic methods, but most of the products will do this by default.

B.2.1 F-Secure Antivirus

Only argument needed is the path to the file to scan. Updates must be applied through automatic updates.

B.2.2 AVG Free

Only argument needed is the path to the file to scan in the form `/SCAN=c:file`. Updating is straightforward.

B.2.3 Bitdefender Free

In addition to the path to the file to scan, the argument `f` is needed to indicate that the object is a file. The program will not scan memory or boot sectors. Updating is done by supplementing the argument `/update`.

B.2.4 Avast Professional

In addition to the path to file to scan, several arguments are needed. `/_` is needed to force a console scan (and not a graphical based one), `/a` will test all files and `/s` turn off sounds. The `/t=A` includes all archive types in the scan. Updating need the arguments `/silent` to disable any user interface, and `/autoreboot` to restart the virtual machine if the updates request so.

B.2.5 Kaspersky Anti-Virus

The argument `SCAN` is needed to indicate a normal scan. `/i0` means that no actions will be taken if an infection is found. Updates must be applied through automatic updates.

B.2.6 Panda

Arguments `-auto` (scan without user intervention), `-nob` (do not scan boot sectors), `-cmp` (scan compressed files), `-nos` (deactivate sounds), `-heu` (enable heuristic scanning methods) are needed in addition to the report file and which file to scan. Updates must be applied through automatic updates.

B.2.7 Trend Micro

Arguments `/nm` (do not scan memory), `/nb` (do not scan boot records) and `/nc[2]` (scanning level include heuristic methods) is needed in addition to the file to scan. Updates must be applied through automatic updates.

B.2.8 McAfee

Arguments `/NOMEM` (do not scan memory), `/REPORT` (report to file) and `/RPTALL` (use heuristic methods and scan compressed files) are needed in addition to the file to scan. Updates must be applied through automatic updates.

B.2.9 ESET NOD32

Needs a separate script to work. The following should be placed in `c:\scan.bat`.

```
del c:\results.txt
cd "c:\program files\eset\eset nod32 antivirus"
ecls.exe --sfx --rtp --adware --unsafe --unwanted --pattern --heur
--adv-heur --action=none --log-file=c:\results.txt c:\scanme
```

The arguments `sfx` (scan archives), `rtp` (scan for runtime packers), `adware` (scan for riskware), `unwanted` (scan for potentially unwanted applications), `-pattern` (scan using normal signatures), `heur`, `adv-heur` (enables advanced heuristics) `action=none` (report only on infection) and `log-file=c: path` (path to log file) are needed as arguments in addition to the path to the file to scan. Updates must be applied through automatic updates.

B.2.10 Avira Antivir

The arguments `-noboot` (do not scan boot sectors), `-nombr` (do not scan master boot record), `-r1` (report only if infection is found), `-ro` (overwrite existing log file) and `-rfc:`

`path` (path to log file) are needed as arguments in addition to the path to the file to scan. The command “< c:

`avira_cmd`

`cr.txt`” is also needed so the execution of the scan will halt and not wait for user interaction. The text file must contain one DOS newline only (carriage return). Updating does not need any arguments, but requires the virtual machine to run non-suspended to work.

B.2.11 ClamWin

Needs separate scripts to work. The following should be placed in `c:\scan.bat`.

```
cd "C:\Program Files\Clamwin\bin"
clamscan.exe --detect-pua --detect-broken --detect-structured
--database="C:\Documents and Settings\All Users\clamwin\db"
c:\scanme > c:\results.txt
```

The arguments `-detect-pua` (detect possible unwanted applications), `-detect-broken` (detect broken executable files), `-detect-structured` (detect structured data such as credit card data or social security numbers) and `-database` (path to the virus signature files) are required arguments in addition to the file to scan.

The following should be added to `c:\update.bat`.

```
cd "c:\program files\clamwin\bin"
freshclam.exe --datadir="C:\Documents and Settings\All
Users\clamwin\db" --config-file="C:\clam\freshclam.conf"
```

The argument `-config-file=c:`
`path` is needed in addition to the `datadir`-argument as in the scan script. The config file is shown below.

```
# URL of server where database updates are to be downloaded from
# If this option is given multiple times, each will be tried in
# the order given until an update is successfully downloaded
DatabaseMirror database.clamav.net

# Number of times to try each mirror before moving to the next one
MaxAttempts 3
```

B.3 Activity diagram

An activity diagram for `vmcom` is drawn to show the program flow in respect to the different operations available from `vmcom`. The diagram follows the UML notation, and each configured antivirus product will venture through the flow shown in the diagram upon execution. The activity diagram is shown in Figure B.1 on the next page, and assumes the following two properties to keep it clean and tidy.

- If `vmcom` would for some reason fail during its execution, actions are taken in respect to `vmcom`'s current state. As an example, if the system reports an error during the step "Log into guest OS", the following steps in the diagram would be skipped, and the virtual machine would be instead immediately suspended. A failure would lead to such immediate suspending of the virtual machine for all cases except if the suspend step in it self fails. If it does, the virtual machine will be instead reverted to the stored clean snapshot. To keep a clean activity diagram, these exceptions are not shown and the diagram assumes a non-failure execution.
- Systems with enough memory do not need to suspend the virtual machines. The diagram however assumes the opposite; that the virtual machines are suspended.

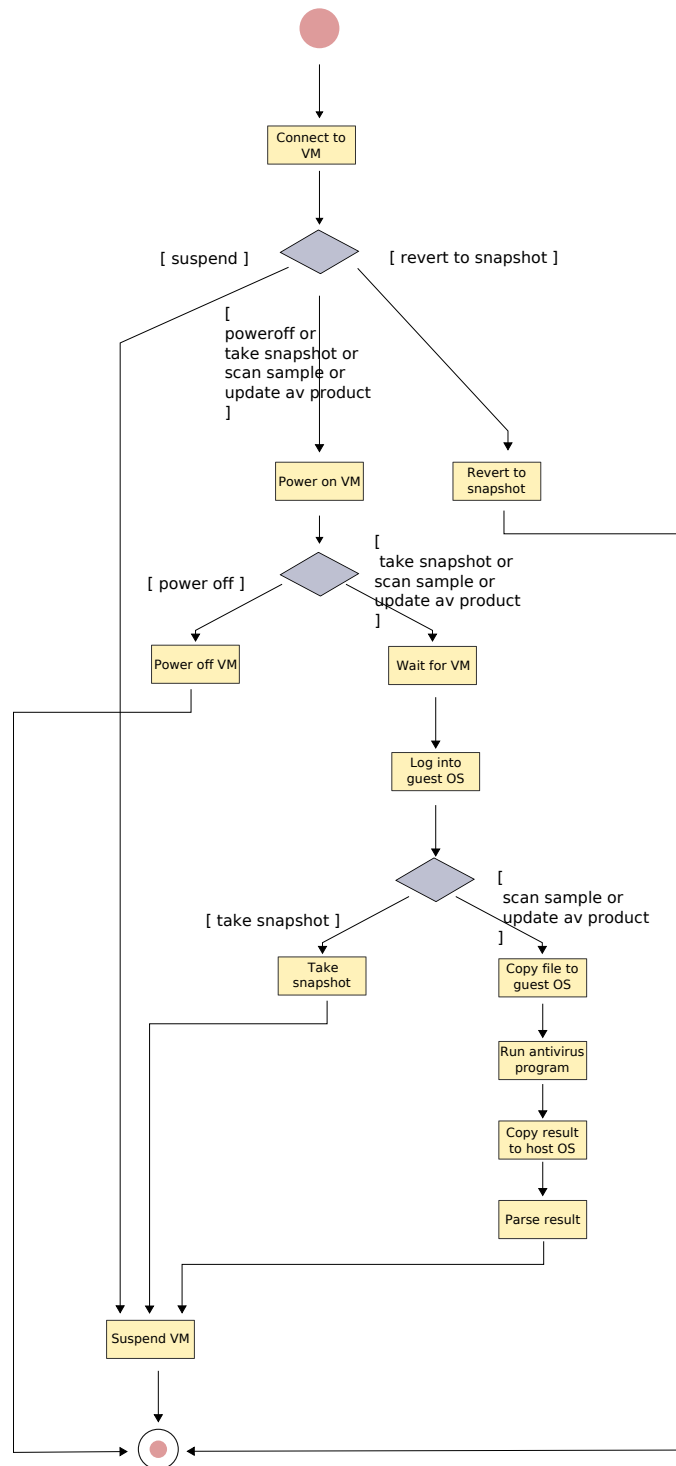


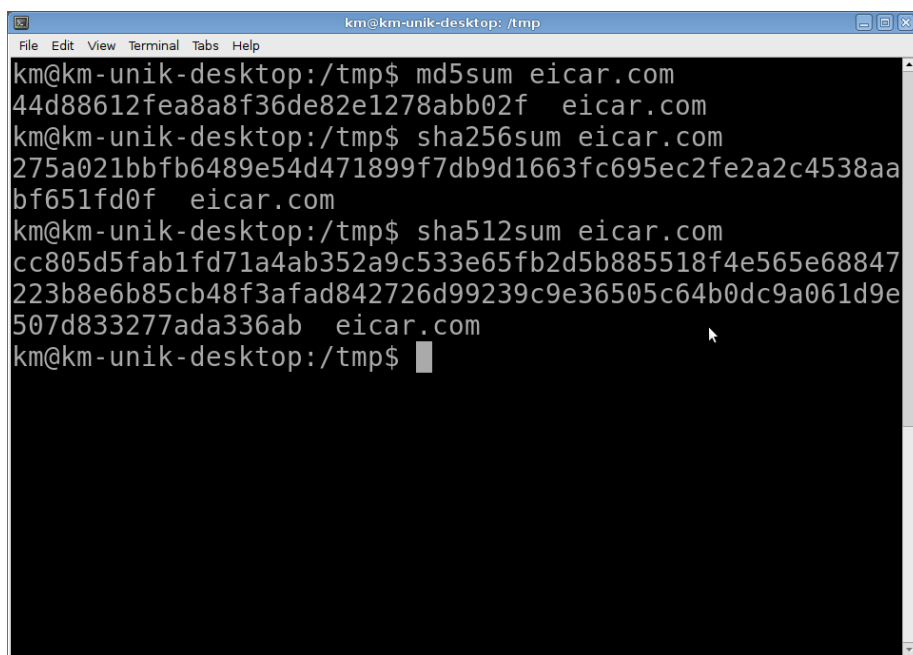
Figure B.1: Activity diagram showing vmcom program flow.

Appendix C

Tool screenshots

This chapter contains screenshots from some of the analysis tools discussed in Chapter 3 and from the `vmcom` program running a file scan.

Figure C.1 shows a screenshot from some of the most common tools for calculating a hash sum. The hash outputs are respectively 128, 256 and 512 bits in size for the three programs shown.



```
km@km-unik-desktop: /tmp
File Edit View Terminal Tabs Help
km@km-unik-desktop:/tmp$ md5sum eicar.com
44d88612fea8a8f36de82e1278abb02f  eicar.com
km@km-unik-desktop:/tmp$ sha256sum eicar.com
275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aa
bf651fd0f  eicar.com
km@km-unik-desktop:/tmp$ sha512sum eicar.com
cc805d5fab1fd71a4ab352a9c533e65fb2d5b885518f4e565e68847
223b8e6b85cb48f3afad842726d99239c9e36505c64b0dc9a061d9e
507d833277ada336ab  eicar.com
km@km-unik-desktop:/tmp$
```

Figure C.1: Screenshot showing usage of tools for three of the most common hash sum methods used during the surface analysis phase.

Figure C.3 shows a screenshot from the network analysis tool, Wireshark and displays parts of the content of a network packet.

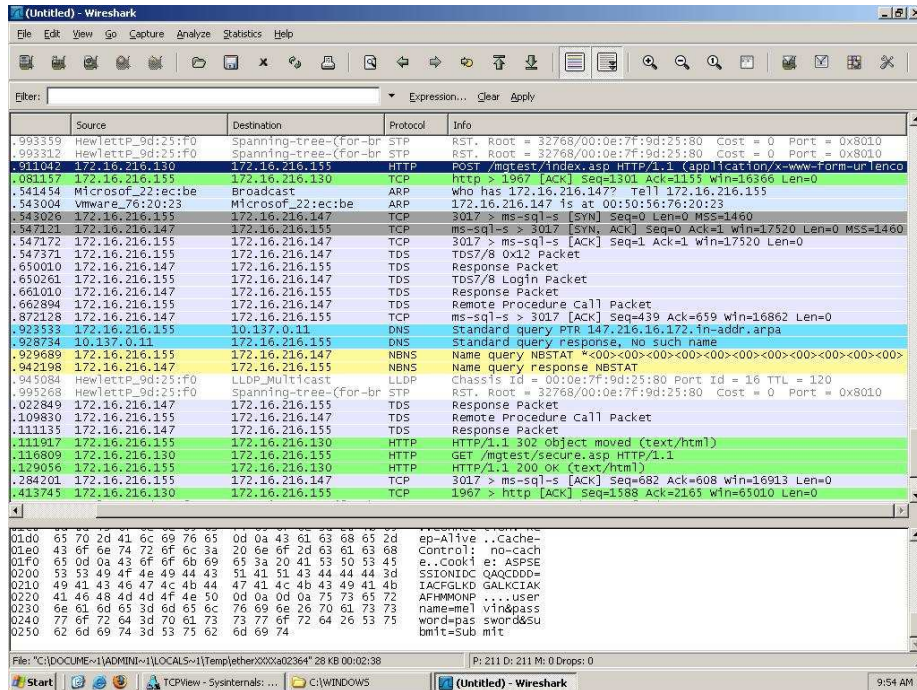


Figure C.3: Screenshot showing usage of the network analysis tool, Wireshark.

Figure C.4 displays a screenshot from the IDA Pro disassembler, which is used as a static analysis tool. The file being worked on is a program for communicating over the SSH and Telnet protocol.

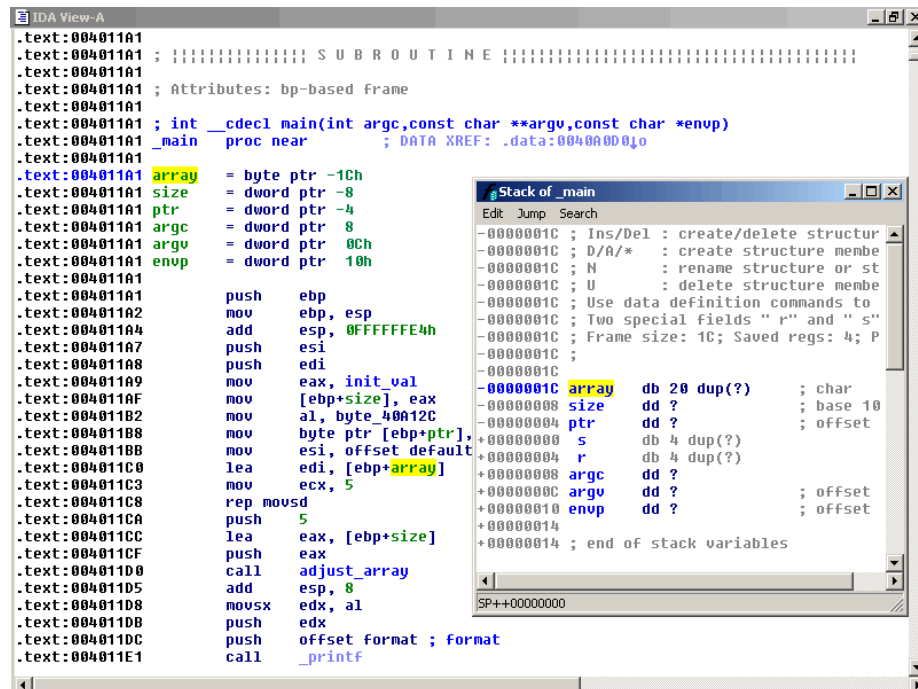
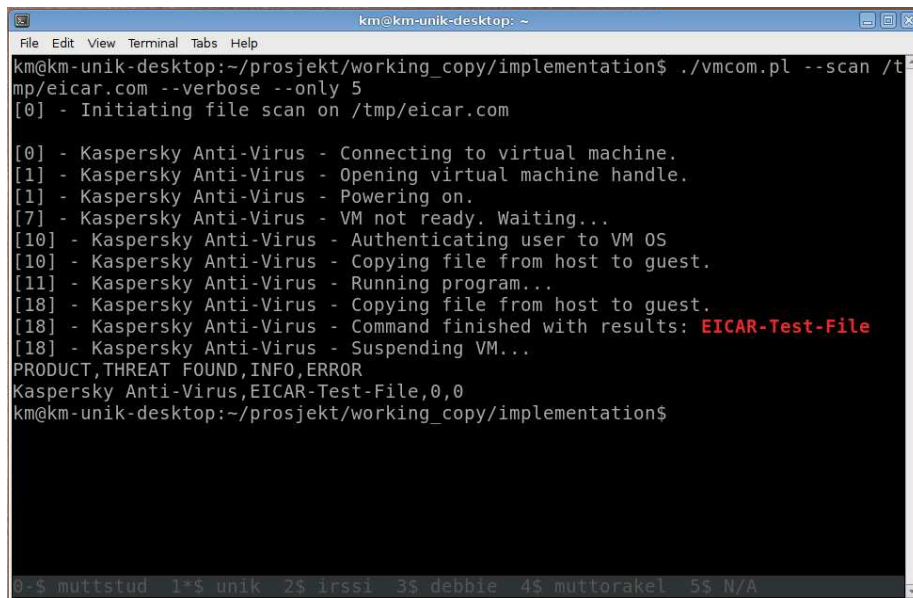


Figure C.4: Screenshot showing usage of the static analysis tool, IDA Pro.

Figure C.5 displays a screenshot from the `vmcom` program running a file scan on the Eicar test file. The file contains a 68 byte long text string. Antivirus applications will consider the file as a virus, and is a good way to test test detection without working with dangerous software.



```
km@km-unik-desktop: ~
File Edit View Terminal Tabs Help
km@km-unik-desktop:~/prosjekt/working_copy/implementation$ ./vmcom.pl --scan /tmp/eicar.com --verbose --only 5
[0] - Initiating file scan on /tmp/eicar.com

[0] - Kaspersky Anti-Virus - Connecting to virtual machine.
[1] - Kaspersky Anti-Virus - Opening virtual machine handle.
[1] - Kaspersky Anti-Virus - Powering on.
[7] - Kaspersky Anti-Virus - VM not ready. Waiting...
[10] - Kaspersky Anti-Virus - Authenticating user to VM OS
[10] - Kaspersky Anti-Virus - Copying file from host to guest.
[11] - Kaspersky Anti-Virus - Running program...
[18] - Kaspersky Anti-Virus - Copying file from host to guest.
[18] - Kaspersky Anti-Virus - Command finished with results: EICAR-Test-File
[18] - Kaspersky Anti-Virus - Suspending VM...
PRODUCT,THREAT FOUND,INFO,ERROR
Kaspersky Anti-Virus,EICAR-Test-File,0,0
km@km-unik-desktop:~/prosjekt/working_copy/implementation$

0-$ muttstud 1*$ unik 2$ irssi 3$ debbie 4$ muttorakel 5$ N/A
```

Figure C.5: Screenshot showing usage of `vmcom`.