Information-based Dependency Matching For Behavioral Malware Analysis

Lars Arne Sand

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2012

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Information-based Dependency Matching For Behavioral Malware Analysis

Lars Arne Sand

2012/07/01

Abstract

Malicious software (malware) has been a constant threat to computer environments. Every year malware inflict staggering amount of damage and incur vast financial losses worldwide. Malware has changed drastically and its purpose, attack vectors and methods are no longer simple. Furthermore the attackers often utilize unknown vulnerabilities, evasion techniques and generator algorithms which drastically increase the impact, effectiveness and quantity of malware. Thus the task falls to security experts to develop tools and techniques to thwart this ever expanding threat. The challenge is to detect all attacks, regardless of evasion techniques, while keeping false alarms to a minimum. This thesis seeks to analyze the application of function call-based malware detection. More specifically function calls with their inter-dependencies, extracted by use of information-based dependency matching. Analysis will be performed to research whether this method is reliable and improve obfuscation resilience. The thesis discusses the difference of performing detection at library call, system call or function call(hybrid) layer, and how well detection can be performed at these layers.

Sammendrag

Skadelig kode som virus, ormer, trojanere har vært en konstant trussel mot datamaskiner og tilhørende nettverk. Hvert år påføres store skader som resulterer i mye arbeid og finansielle tap. Skadelig kode har endret seg drastisk, og dets formål, angreptsvektor og metode er ikke lenger enkel. Videre benytter angripere ofte ukjente sårbarheter, beskyttelsesmekanismer og algoritmer som drastisk øker omfanget, effektiviteten og mengden skadelig kode. Oppgaven faller derfor til sikkerhetseksperter for å utvikle verktøy og sikkerhetsmekanismer som kan avverge og motvirke denne evig økende trusselen. Utfordringen er å oppdage og stoppe alle angrep, uavhengig av hvilke unnvikelsesmanøvre og metoder som benyttes. Denne oppgaven forsøker å detektere skadelig kode basert på API-kall analyse. Mer spesifikt, så benyttes API-kall og avhengigheter mellom disse. Disse avhengighetene opprettes ved hjelp av informasjon fra API-kallene. Oppgaven går så ut på å undersøke om man kan opprette avhengigheter mellom API-kall på en pålitelig måte. Videre undersøkes det om denne type deteksjon er mer robust mot typiske unnvikelsesmanøvre (obfuscation techniques). Til slutt så analyseres forskjeller i deteksjon for brukermodus og systemmodus. Dette være seg forskjellen på vanlige og systemkritiske oppgaver, og om det finnes forskjeller mellom disse for deteksjon av skadelig kode.

Acknowledgments

I would like to thank my supervisor, Prof. Katrin Franke, for providing excellent guidance and assistance throughout the project. During the thesis she has shown great interest in my work and always been able to provide insightful discussions, which has guided this thesis.

Secondly I would like to thank my classmates at GUC, for discussions, feedback and motivation during the writing of this thesis. Especially so, I would thank my opponent Andreas Tellefsen for his support and help, as well as Svein Roger Engen for expert knowledge of the Linux kernel.

Finally I would like to thank my family for motivation and support throughout my studies. As well as my girlfriend Shirley Chiang, for her understanding, support and patience.

Contents

List of Figures

List of Tables

Glossary

Control Flow Graphs Construct and traverse node with regard to node content and inter-relationship

Decompressor tools Tools for reversing data compression and analysis of portable executable files

Deobfuscator A program whose purpose is to implement data-flow analysis algorithms to automatically remove irrelevant functions [7]

Disassembly Is the process of translating machine code into assembly language

False positive Incorrectly classified as malicious data

False negatives Incorrectly classified as benign data

Intrusion Prevention System A system whose task is to report and prevent any malicious activity

Metamorphic malware Evade detection by obfuscating the entire virus [8]

N-gram Collection of n-byte strings which are typically used for detection based on statistical analysis or learning

Non-Sequential Disregard function call order. Simply counts occurrences

No free lunch Whenever one have a classifier that solve a problem. One can always create a problem where the classifier fails

Obfuscation Any technique used to evade detection

Packers Software applications that store either encrypted or compressed executables (packed), in such a way that when executed the packed executable is loaded into memory and executed [3]

Polymorphic malware Evade detection by obfuscating decryption algorithm [8]

Sequential Take sequence of function calls into account

Tainting The process of marking data and monitoring and checking its flow throughout the system

True positive Correctly classified as malicious data

True negative Correctly classified benign data

Abbreviations

AV Anti-Virus

FP False Positive

FN False Negative

IDS Intrusion Detection System

IPS Intrusion Prevention System

MBR Master Boot Record

NOP No Operation Pointer

OEP Original Entry Point

OS Operating System

OWASP The Open Web Application Security Project

PE Portable Executable

PII Personal Identifiable Information

TN True Negative

TP True Positive

TSR Virus Terminate and stay resident virus

1 Introduction

The purpose of this chapter is to give an introduction of the subject and challenge in question, as well as justification and motivation of its importance. The chapter also proposes research questions to guide the thesis, and a discussion of the planned contributions.

1.1 Topic

Malware is short for malicious software and can be defined as any program or file that is harmful to a computer environment or its user [9]. Malware has existed since before the modern and wide spread use of computers. Historically speaking, one of the first samples of malware was a virus which infected the game ANIMAL in 1975 [1]. The purpose of the virus was simple, as it copied itself to every directory [10]. As malware developed, the intrusion detection systems (IDS) were created. The role of an IDS is to detect and report any malicious behavior. The problem however is that malware, like computers and software has changed drastically. The purpose, attack vectors and methods are no longer simple. Furthermore the attackers often utilize unknown vulnerabilities, evasion techniques and generator algorithms which drastically increase the impact, effectiveness and quantity of malware. Thus the challenge of any IDS is to detect all attacks, regardless of evasion techniques, while keeping false alarms to a minimum.

1.2 Keywords

Malware, intrusion detection system, behavioral-based detection, machine learning, function call analysis, dynamic API traces.

1.3 Problem description

Malware detection by itself is a relatively new field of science. Regardless, it has always been an important part- and heavily researched subject of information security. Some key reasons for this is the large amount of money and opportunities to be gained by controlling information systems. Because of this, large investments by organizations, governments and criminals alike are made to increase cyber-warfare capabilities. The result is ever increasing threats, while the security is lagging behind.

There exist more advanced methods of malware detection, which utilize statistical methods, clustering or learning. However, these algorithms often have high false positive rates or low detection accuracy. Due to these downsides, they are seldom deployed. Because of this, typical malware detection utilizes signature matching, since these systems are accurate and provide low false alarm rates. Furthermore users have high expectations for both reliability and speed, such that security measures which result in high overhead are rarely acceptable. The attackers spend much resources on finding new attacks as well as employing cutting-edge obfuscation techniques to evade detection. Previous work countered this by bringing detection to the lower levels of computer hierarchy, namely by function call analysis. Function call analysis can be performed both statically and dynamically. Static analysis is typically performed through source code analysis or disassembly,

while dynamic analysis is done through function call tracing. This thesis focuses on the latter.

There exist several ways of analyzing function call traces. Both sequential, non-sequential, use of arguments, resource use, n-gram, tainting etc. These are further discussed and explained in Section 3.2. This thesis focus on function calls and their inter-dependencies and thus sequences of function calls are used. The inter-dependencies are created using information about function call parameters and return values. Because of this we call this method information-based dependency matching. Furthermore it will analyze at what level function call analysis is best performed. Be it at a high level for program libraries or low level near system kernel. The goal is to gain a better understanding of pros and cons with regard to detection accuracy, obfuscation and throughput at the different layers.

1.4 Justification, motivation and benefits

The importance of computer systems in todays society cannot be underestimated. Computer systems are not only used by the common man, but also by critical infrastructure such as water, sewage, power, communications, health-care and day-to-day infrastructure such as ordering and deliverance of food. It is therefore essential to research methods which safeguard the integrity, availability and confidentiality of any such system.

Dynamic function call-based malware detection should be able to better withstand obfuscation attacks. Furthermore by looking for anomalous data, the system would be able to detect previously unknown attacks. Consequently, a successful implementation of this method could improve obfuscation resilience and increase its ability to detect novel attacks.

1.5 Research questions

This thesis is to a large extent motivated by the use of information-based dependency matching for malware detection and its advantages with regard to obfuscation resilience. At the same time however, information-based dependency matching has known weaknesses [11, 12] that needs further studies. Subsequently it will compare the difference of performing function call detection in user mode (library calls) and kernel mode (system calls). In particular, this thesis seeks to answer the following questions:

1. Is the use of information-based dependency matching reliable?
2. Will obfuscation resilience increase by use of information-based dependency matching?
3. Which features provide the best accuracy with regard to false positives and detection rate for the different layers?

1.6 Contributions

This master thesis seeks to provide a better understanding of the abilities of information-based dependency matching. Especially with regard to obfuscation resilience and reliability. The thesis also discusses the differences of performing detection at different function call layers. This has to the author's knowledge not been discussed previously and is important as it might have a direct impact on the method's detection rate, obfuscation resilience and throughput.

1.7 Thesis outline

This section provide a brief summary listing of the content presented in this thesis. The listing is based on the chapters, where each chapter and its content is described. First the literature about malware, related work and graph matching is presented. Then the methodology is presented, followed by the results of the analysis as well as discussions, conclusion and further work.

- Chapter 2 presents literature related to malware detection. First the malware taxonomy is described. Followed by literature about malware detection and intrusion detection methods and their respective categorizations. Finally there is a discussion on the different malware obfuscation techniques.
- Chapter 3 presents the related literature of the thesis. First it describes function call analysis and the differences of user mode and kernel mode detection. Then the state of the art with regard to malware detection for function calls is outlined. Finally an introduction to information-based dependencies is given. This section discuss how one can create dependencies between function calls.
- Chapter 4 provides an introduction to the graph matching problem. First it discusses what graphs are, then the different graph matching methods. The chapter finalize with a section on graph-based learning, and how this is implemented in SUBDUE.
- Chapter 5 presents the methodology and covers all aspects of how the thesis is carried out. This includes the generation of dataset, to pre-processing and choice of detection classifier, as well as the methods required to answer the research questions. The methodology is chosen based on the related literature presented in Chapter 3 and 4.
- Chapter 6 include all the experimental results for each research question. First the reliability testing of information-based dependencies is performed. Then the obfuscation resilience of information-based dependency matching is analyzed. Finally the detection rates of information-based dependency graph matching by use of SUBDUE are evaluated.
- Chapter 7 provides a discussion of the theoretical considerations and practical implications as well as summary of the thesis.
- Chapter 8 concludes and summarizes the most important findings in this thesis
- Chapter 9 presents a range of topics that should be further analyzed to better understand the inherent capabilities and implications of the method.

2 Malware detection

This chapter provide an in depth discussion of the malware taxonomy. It then discusses a series of malware detection and intrusion detection methods and their respective categorizations. Finally there is a discussion on the different malware obfuscation techniques.

2.1 Malware Taxonomy

There exist several definitions for malware. Although mostly similar they do have differences worth mentioning. For instance Preda et al. [13] define malware as *a program with malicious intent that has the potential to harm the machine on which it executes or the network over which it communicates*. This definition cover both malicious actions performed toward the machine and the network. What it ignores however is the user of this system. This is important as malware does not only damage the system, but also invades the users' privacy by accessing and changing documents and data. Subsequently the following definition will be used: *malware is short for malicious software and can be defined as any program or file that is harmful to a computer environment or its user* [9].

As malware is a collective term for malicious software, there exist several sub-categories of malicious software. These terms are discussed below.

2.1.1 Virus

A Virus is a program that inserts itself into one or more files and then performs some action [2]. One of the key characteristics of a virus is that it needs a host to propagate further. Be it a file transferred over a network, USB flash drive or a shared network device. Bishop [2] identifies insertion phase and execution phase as the two key phases of a computer virus. In the first phase the virus inserts itself into another file and in the second phase it executes. Over the last couple of decades several different categories of viruses has been identified. These are:

Boot sector virus

A boot sector is the part of the hard drive that contain data which bootstrap the system or mount disk drives [2], typically called the Master Boot Record. A virus that infects the MBR is called a boot sector virus. Since the content of this sector is executed first, the boot sector virus is able to start alongside the operating system. A method commonly utilized by boot sector viruses is to overwrite the MBR (512B) with the virus itself, then copy the original MBR to succeed the virus in such a way that the virus can point to the original MBR and control the execution [1]. Example is provided in Figure 1. Brain virus is an example of a boot sector virus. It is one of the oldest viruses and was detected in January 1986 [14].

Executable virus

An executable virus is a virus that infects executable programs [2]. The common method of executable viruses is to insert itself after the file header. Thus the virus will be executed before the rest of the program. This method is displayed in Figure 2. The Jerusalem Virus is an example of an executable virus [15].

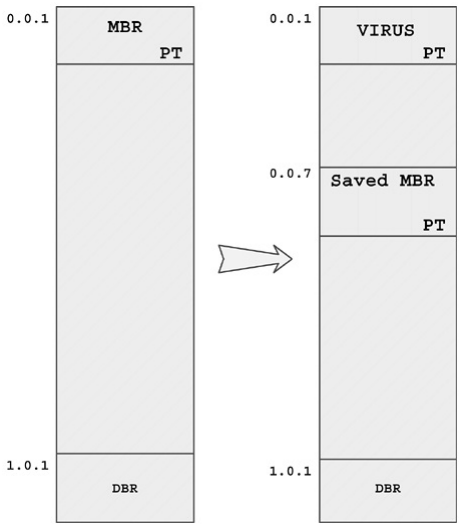


Figure 1: Boot Sector Virus [1]

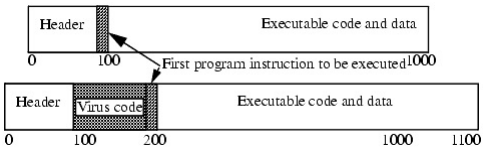


Figure 2: Executable Virus [2]

TSR virus

Terminate and stay resident virus [2]. These are viruses that remain resident in memory after application has terminated. For instance after boot up the brain virus (boot sector virus) is loaded into memory, instead of being run once at boot [2].

Stealth virus

Stealth viruses seek to hide themselves by concealing infected files and processes. According to Bishop [2] a stealth virus conceals infected files, while Szor [1] state that stealth viruses always intercept functions, such that the user receives modified and subverted data. As such, it inhibits rootkit like behavior. Rootkits are discussed further in Section 2.1.3. Viruses utilize these concealing methods to various degree, as for instance the boot sector Brain Virus was programmed to show the user the original MBR [1].

Encrypted virus

Encrypted viruses consist of an enciphered virus body along with a small decipherment routine [2]. Malware detection software quickly realized that viruses had distinct signatures. Thus they implemented simple signature-based detection schemes. The malware authors countered this by enciphering the virus. This made the virus itself much harder to analyze, as one would have to decrypt in order to analyze. One can however run the virus and analyze dynamically as well as extract the virus itself from memory. Hence the encryption only provides an additional layer for the analyst. Signature-based detection is equally efficient, as the malware detection routine can identify the decipherment routine of the virus. One of the first viruses that utilized encrypted was the Cascade virus [16, 1]

on DOS. The virus body in Cascade was similar to that shown in Figure 3, where the decipherment routine is located in front of the enciphered virus.

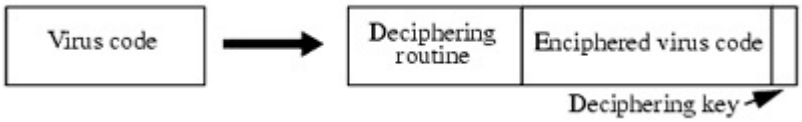


Figure 3: Encrypted virus [2]

Polymorphic virus

A polymorphic virus changes its form each time it inserts itself into a new program [2]. Polymorphic were the result of the next step in the evolution of virus after encrypted viruses. Since malware detection software was able to identify the decipherment routine and successfully detect encrypted malware, the malware authors sought to make malware which would change on each infection. Subsequently this is an attempt to fool the signature-based systems. Furthermore it is often used in conjunction with encryption-based viruses [2]. One of the first examples of polymorphic viruses was the 1260 Virus, which was written by Mark Washburn in 1990 [1]. The 1260 Virus utilized junk insertion which in turn changed the size of the virus. Thus the signatures of the *known bad*, changed as well. Since they no longer could look for simple signatures.

Macro virus

A macro virus is a virus composed of a sequence of instructions that is interpreted, rather than executed directly [1]. Except from the fact that macro viruses go through an interpreter they are no different than regular viruses. The virus may be written in any macro language, as for instance MS Word. One of the best known macro viruses are the Melissa virus [17] which infected Word 97 and 98 on MS Windows and Mac OS X [2].

2.1.2 Worm

A worm is defined as a program that copies itself from one computer to another [2]. The key difference between a worm and a virus is the fact that the virus needs a host to propagate. The worm copies and sends itself to computers via available networks, network shares, e-mails etc. One of the first known malicious worms was the Morris worm which hit the Internet in November 1988 [18]. Named after its creator Robert Morris Jr., the Morris worm propagated fast throughout the Internet. Since then advanced worm propagation techniques such as hit-list scanning have been researched [19]. Hit-list scanning is a technique which creates an initial list of vulnerable targets before starting the infection. Thus when infection takes place the hit-list is transferred to the infected clients, which in turn target parts of the hit-list to increase propagation [19]. The effect of this technique is a drastically increased propagation throughout the network.

2.1.3 Rootkit

Rootkits are defined as a combination of malicious software that provide backdoor access to the machine as well as modifications which hide the rootkit and traces of intrusion [20]. Rootkits are further categorized into the following four categories:

- Application-layer rootkits
- Library-layer rootkits
- Operating System-layer rootkits
- Hardware-layer rootkits

These subcategories are discussed below.

Application-layer rootkit

Application-layer rootkits, also called command-layer rootkits target the higher layers of the computer architecture. Be it files, registry or other application specific artifacts on a host. Typical actions include deleting logs and hiding files which makes the malware inconspicuous. The downside however is that application-layer rootkits doesn't subvert the system kernel. As such they are quite easy to detect, since one can find them using common signature-based systems.

Library-layer rootkit

Library-layer rootkits goes one step further than application-layer rootkits, since they manipulate the libraries used by the applications. Subsequently they are able to subvert a set of applications which use a specific library. One can for instance manipulate libraries of cryptographic hash functions to return benign values, such that a known bad file is never found. From a forensic perspective this can be countered by utilizing external tools when analyzing a computer. For instance by loading a USB-flash drive which has standalone software which doesn't depend on host DLL files.

Kernel-layer rootkit

The third method of system subversion is called kernel-layer rootkits. Kernel-layer rootkits focus on compromising the kernel of the operating system. This includes methods such as in-line hooking of system call tables [21, 22, 23] or direct-kernel-object manipulation [24].

Hardware-layer rootkit

Hardware rootkits focus on firmware and hardware subversion. These rootkits are immensely complex and very challenging to detect as one rarely test such equipment for malicious activity. A taxonomy of hardware trojans is provided in [25] by Tellefsen. By definition hardware trojans and hardware-layer rootkits are not the same. However, oftentimes they exhibit similar behavior. For instance a GPU which is programmed by the manufacturer to leak sensitive information and hide its tracks is both a rootkit and a trojan.

2.1.4 Backdoor

A backdoor is a mechanism which bypasses normal security checks [18]. As such it might be a part of software or software configurations. Typically when a hacker gets access to a computer he seeks to install a backdoor which allow for easy return. Hence it is not a malware that propagates, such as viruses and worms, but rather a method and technique commonly utilized by computer criminals. A typical backdoor opens a port which allows a perpetrator to connect to the client.

2.1.5 Trojan

A trojan is a program with an overt (documented or known) effect and a covert (undocumented or unexpected) effect [2]. Malware is typically concatenated to regular software and posted on file sharing networks. As such, trojans are one of the most prominent threats in today's Internet-based community. Previous research on malware in peer-to-peer networks found that a total of 68% of all executables and archives contained malware [26]. Hence the likelihood for infection for an unsuspecting user is very high.

2.1.6 Spyware

Aycock [18] defines spyware as software which collects information from a computer and transmits it to someone else. This is a broad definition, in which case most software is spyware, which is not true. Hence, in this thesis we define spyware as software which collects and transmits information from a computer without the user's knowledge or consent. Typically the goal of spyware is to collect personal identifiable information (PII) such as name, e-mail address, credit information and passwords. This is then used to access the user's accounts and purchase items in the user's name or extract his or hers funds.

2.1.7 Adware

Adware is similar to spyware, except it usually gathers information about the users and their habits [18]. It then feed the user pop-up advertisements or redirect browsing to special web-sites in order to entice the user into buying products.

2.2 Malware detection and intrusion detection

Malware detection is defined as a system that attempts to determine whether a program has malicious intent [8]. While intrusion detection is defined as the *process of monitoring the events occurring in a computer system or network and analyzing them for signs of possible incidents, which are violations or imminent threats of violation of computer security policies, acceptable use policies, or standard security practices* [27].

Malware detection is similar to intrusion detection systems, since they both try to identify malicious actions. Key difference being that malware detection only focus on malware, I.E. malicious software, while IDS systems seek to identify all possible incidents or malicious actions. As such malware detection might be seen as a subcategory of IDS. Due to this fact, most of the terminologies of IDS stand true for malware detection. Hence, to provide an introduction in malware detection we must also discuss the general aspects of IDS.

Intrusion detection systems are typically categorized after scope of protection, model and analysis method. A brief introduction to these categorizations is provided below.

2.3 Scope of protection

By scope of protection one check whether the IDS is network-based, host-based or application-based [28]. It is a way to categorize which aspects of the system the IDS monitor.

- Network-based: By network-based, the IDS monitors all traffic sent over the network. The type of network traffic depends on the sensor location within the network.
- Host-based: Host-based IDS on the other hand monitor all actions performed on a

host. Scope depends on algorithm and implementation, but typical monitored data include registry, files, logs, function calls and network data.

- Application-based: Application-based IDS are as the name imply specific for each application. These IDS's monitor data internal to each application such as application events, logs and data. It might also be inspecting protocol specific information. Typical example would be SQL monitoring for a database system.

This thesis will largely focus on host-based in general, as it focuses on function calls, which represent all activity and not just application specific data.

2.4 Scope of model

By model one categorize between anomaly- and signature-based [28, 29]. The difference in these methods is how the detection is performed. Either by looking at the known-good, then creating profiles and classifying everything else as anomalies (anomaly-based). Or looking for known-bad signatures, then classifying everything else as benign (signature-based).

2.4.1 Signature-based

Signature-based detection looks for the known bad. As opposed to the first generation antivirus scanners this doesn't only mean known-bad cryptographic hash sums. However, any malicious characterization or statistical properties, be it information such as variables and URLs or applications, protocols and files. The key downside to signature-based detection is that since it looks for the known bad, it sometimes creates too specific signatures. The outcome is that by performing small modifications to the malware a perpetrator can easily avoid detection. Previous studies has in fact proved that signature based detection is vulnerable to obfuscation techniques such as polymorphism and metamorphism [30]. The challenge of signature-based detection is to change the signatures, such that they are more general, and able to detect modified malware categories, but not so general that they create false positives.

2.4.2 Anomaly-based

Anomaly-based detection focus on identifying the known-good. As a result it has an advantageous ability to detect new malware, such as mutations of an existing malware family, but also novel attacks and techniques. The reason that anomaly-based detection is more suited for detecting new attacks is because any deviation from the known good is classified as malicious. This however has a downside, since it often cause a large set of false positives. Hence in many ways anomaly-based detection is the opposite of signature-based detection. This also holds true for the challenges, as one have to create more specific templates, which in turn reduce the amount of false positives.

2.5 Analysis method

By analysis method one check whether detection is performed pre-execution or during execution, I.E. static or dynamic [29], respectively. Common methods of static analysis is source code analysis or disassembly, while dynamic methods execute, then trace or log execution flow, or look at changes in the system post-execution.

Analysis method is especially important in malware detection since it depends on

whether the system needs to execute the malware or not. The safest option of course is to analyze the file pre-execution. However, sometimes malware is obfuscated in such a way that it cannot be analyzed statically. These methods will be discussed next in Section 2.6. As a result, one has to execute the binary and investigate the artifacts it creates in the system. These artifacts might not be written to disk, but reside in memory only. Furthermore one typically has a large set of artifacts to uncover. Artifacts may include, but not restricted to memory data, disk data such as files, registry, folders, file metadata, binaries, logs etc. To make matters worse malware typically seek to erase their tracks. Thus it is important to note where detection takes place, whether one trace every action or simply look at the artifacts post-execution.

Another key challenge is that the system itself generates a lot of artifacts. This is beneficial in some cases as one can detect malicious actions. However, it also means that a lot of the artifacts one investigate might be generated by the system itself, and not a result of the malicious file. This too is important to consider when implementing the analysis method. For instance a dynamic malware detection method based on comparing images pre- and post-execution will be susceptible to a lot of benign system artifacts. A dynamic malware detection method that traces a process on the other hand, will only see that process's actions. A potential downside however might be that the process spawns additional threads and processes that the detection method isn't able to follow. Hence the malware might be able to break free and avoid detection. Although two simple examples, they show the complexity of computer systems that is needed to consider when implementing an analysis method.

2.5.1 Static analysis

Static analysis is a method used to determine the inner workings of software without actually executing the program. This is typically done through, disassembly, signature scanners and decompiling. The key advantage of static analysis is the fact that one doesn't need to execute malware.

First-Generation Scanners

Scanners are the word that comes to mind when one speaks of Anti-virus(AV) software. However these AV's utilize a range of different techniques to detect malware. The first-generation scanners is by far the simplest, which utilize techniques such as string matching [1]. This is basically an exact string search for some known bad string. These were then extended to support wildcards. That is, the strings may include unknown data for the size of the wildcard (typically one byte) [1]. A very simple example of a string matching is provided in Figure 4. As one can see in the figure the matching algorithm matches the string character by character. If the current character matches it checks the next one. Because of this methodology, similar strings will have a series of matches before the mismatch occurs, which result in performance degradation. To counter this, the first-generation signature scanners started using cryptographic hash-values [1].

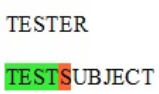


Figure 4: First generation string matching

The AV developers further sought to make AV faster and reduce the amount of disk access. Thus they implemented top-and-tail scanning, which basically is to scan the header and the tail of the files, instead of the whole body [1]. Entry-point and fixed-point scanning was then implemented. These techniques made AV scanners even faster, as they take advantage of the fact that most files have entry points for objects, and that many viruses target these entry locations [1].

Second-Generation Scanners

As malware evolved and started mutating using no instruction pointers (NOP) [1], which successfully avoided the first-generation scanners. The AV scanners had to change accordingly and evolved into the second-generation scanners. Which started using smart scanners, that ignored such NOP [1]. A hybrid method called skeleton detection was also implemented. This method dropped all NOP, white space and non-essential instructions, such that only the essential code is parsed. Thus increasing the AV's ability to detect malware of the same family, as mutations were less successful [1].

Second-generation scanners then started utilizing nearly exact identification, which is the use of two checksums for only parts of the virus body. If one of the checksums matches a warning of a potential match is given [1]. This was succeeded by the exact identification, which instead utilized checksums for the whole virus body which was constant [1]. I.E. variables were excluded to remove environment-dependent data. Such that only the *constant virus body* is used as a signature. Exact signature matching has the advantage that it is able to differentiate between malware hybrids and different variants of the same family [1]. However, its exact matching is likely to make malware obfuscation techniques more effective.

Malware-Specific Scanners

Malware-specific scanners are the next step in the evolution of AVs. Malware-specific scanners search, as the name imply for specific malware [1]. It's more commonly known as algorithmic scanning, however we feel this name is more suitable as it actually reflect the technique's intention. Malware-specific scanners have had several issues with portability and stability [1]. However implementations of abstract programming languages such as Java-like portable code has given mitigated the portability issues as they can run on any system [1]. Since malware-specific scanners typically are processor intensive, they need to filter their search. For instance, searching for executable viruses can be limited to executable files. This can of course be exploited and avoided using obfuscation. However, this is further discussed in Section 2.6.

Source code analysis

Source code analysis is a method commonly affiliated with programming quality assurance. However analyzing source code manually can be an efficient way to detect malware as well. In the right hands, access to the source code can reveal the malicious program's intentions, the attackers proficiency as well as information of how the intruder got access and which vulnerabilities were exploited. Information of source code analysis tools and techniques is provided on The Open Web Application Security Project (OWASP) [31].

Disassembly

Disassembly is the process of taking an executable binary as input and generate the assembly language representation of the program as output [7]. The program, which is

represented by machine code (zeroes and ones) is interpreted by the disassembler, but instead of executing code, it stores the textual representation in assembly [7]. Disassembly is usually performed by disassemblers, I.E. programs specially designed for this task. These disassemblers typically support multiple CPU architectures, as disassembly is a processor-specific task [7].

Decompiling

Decompilers are the next step in the evolution after disassemblers. Disassemblers convert machine code into human readable assembly code. The challenge is that for most humans, assembly code isn't easy to interpret. Thus the decompilers translate machine code into a high level language (c-like code) [7], which is far easier to interpret. This typically results in something similar to the actual source code. It is of course not completely similar to the source code, however it is likely to be helpful when analyzing large segments of code.

2.5.2 Dynamic analysis

Dynamic analysis is defined as the process of executing malware in a monitored environment to observe its behaviors [32]. As opposed to static analysis, one actually executes the malicious code, which can be dangerous unless performed securely. There are several ways of performing dynamic analysis. These include taking snapshots and comparing pre- and post-execution and analyzing during execution by use surveillance tools or debuggers

Dynamic analysis can be performed on two different architectures. That is, physical and virtual hosts respectively. On physical hosts you control the computers using re-imaging software, while the virtual hosts are controlled by virtualization software. Both the virtual and the physical host computers are typically controlled by a controller [32]. This is a server that decides which malware to load and analyze, and when to re-image or revert snapshots.

The advantage of physical hosts is that they behave as regular hosts would. As such virtualization-aware malware will have no effect. The unfortunate downside however is that the process of re-imaging is slower than reverting snapshots [32].

Virtual hosts on the other hand are quick and easy to set up. Furthermore one can use snapshots to control the host environment, as well as using several different baselines. A baseline represents the operating system and its software configuration. As malware might behave differently for different operating system (OS) versions, it is important to test the most common ones, to which are vulnerable.

Securing the environment

There are two downsides of this virtualized approach. First, malware can break out of the environment and onto the host computer [33]. Second, malware doesn't behave as it usually does when virtualized, as it won't be able to communicate with command and control servers or outside environment. The first issue can be mitigated by:

- updating host computer and virtualization software
- dedicated host computer which is not connected other networks
- monitor host with integrity detection systems

The second issue we can control by selecting malware that i) doesn't need interaction with outside network, ii) enable by emulation, I.E. services such as DNS and e-mail services can be emulated on the virtualized host to enable the malware and its functionality. This is a common method of dynamic reverse engineering [34].

In addition to these precautions it is common sense to isolate the network for both physical and virtual lab environments, given that malware' propagation abilities.

Pre- and post-execution comparison

In this method one compare chosen data pre- and post-execution. For instance by taking a snapshot of the registry, file system or memory, then run malware and observe its behavior, by checking which changes occurred on the system. Regshot [35] is one example of such a program, which analyze changes made in registry.

Run-time behavioral analysis

This method is common in live forensic analysis where one need to analyze malware or running processes on a live system. It is based on using tools to analyze artifacts on a running systems, such as memory, processes, handles, threads, files, network connections, connected devices and of course the file system. A comprehensive software analysis suite commonly used for this purpose is the Windows Sysinternals [36].

Debugging

A debugger is a program that run and monitor the execution of other programs [37]. While debugging is the act of locating bugs in software [37]. Debugging is oftentimes not classified as dynamic analysis [32]. However, due to the fact that by definition dynamic analysis executes malware, and the debugger is a program that controls this execution we have classified it as such.

A debugger can be a powerful tool to find vulnerabilities and analyze malware [32]. One can start debugging a new process or even hook into running processes. This makes the debugger a powerful tool for dynamic analysis.

2.6 Obfuscation techniques

There are few good and proper definitions of the word obfuscation. One paper [38] defines obfuscation as a technique that makes programs harder to understand. However, obfuscation is not only meant to trick the human analyst, but also techniques that throw technical systems off track. Such as polymorphic malware for signature-based detection. Thus in this thesis we define obfuscation as any technique used to evade detection. It is important to note that most obfuscation techniques seek to change a program, in order to avoid detection, but keeping the same functionality.

Obfuscation can be performed both manually and automatically. By manual obfuscation we think of methods that are implemented during the programming, while automatic obfuscation is implemented by a program. Automatic obfuscation is most often advantageous as it provides better mechanisms for automatically obfuscating the complete program [7]. Furthermore it usually obfuscate after the program is compiled, such that one doesn't make the source code less readable for the developers [7].

Obfuscation is not only used in malware, but actually a common protection mechanism to protect software from competitors. In fact one of the first surveys on obfuscation methods discusses obfuscation techniques for software protection [39]. The survey discusses a range of obfuscation categories such as data obfuscation, layout obfuscation,

control obfuscation and preventive transformations. Within these subcategories a total of 26 obfuscation techniques are discussed and analyzed with regard to a set of proposed metrics. These are potency, resilience and execution cost. It is important to note that even though many of the same obfuscation techniques are used for both software and malware, their efficiency may differ, as they have different requirements. For instance software has performance criteria, while malware need to change continuously in order to beat signature-based detection systems [38]. Another application worth mentioning is the use of obfuscation for security. As many hackers seek to analyze software and websites in order to find vulnerabilities, for example poor input validation etc., obfuscation can be used as a security measure to counter such probing and analysis [38].

In this thesis we will not go in depth on obfuscation and measure the quality of the different techniques. Nor will we discuss obfuscation with regard to software protection or security. But rather discuss the existing obfuscation techniques and how they evade today's malware detection systems.

2.6.1 Compression & Encryption

Malware and malware detection has often been discussed as a cat and mouse game. This is not without reason, as the malware authors have always created some malicious code or obfuscation technique, and when the malware detection catch up, they create a new one. Malware started out simple, without any forms for obfuscation. Then, when malware detection started using signatures, the malware authors had to adapt and hide their code. This was performed using compression and encryption algorithms. Encrypted viruses were discussed earlier in Section 2.1.1. Due to this reason we will not go in depth in this section, but rather describe how encryption work as an obfuscation method in general as well as some methods of enhancement.

Encryption and compression is implemented in such a way that the malware consist of an encrypted part and a decryptor. The decryptor is a sequence of code responsible for deciphering the enciphered malware [40].

Since encryption is one of the oldest obfuscation methods, there has been developed several ways of enhancing its efficiency. The following techniques was proposed in [1]:

- Change direction of the encryption/decryption loop, such that both forward and backward loops are supported [1]
- Use multiple layers of encryption. Such that the first decyrptor decrypts the second one, while the second decrypt the third, and so on [1]. An example of this method can be viewed in Figure 5
- Several encryption loops take place one after another, with randomly selected forward and backward loops [1]
- Use of more than two keys to decrypt each encrypted piece of information, combined with long keys [1]
- Nonlinear decryption [1]

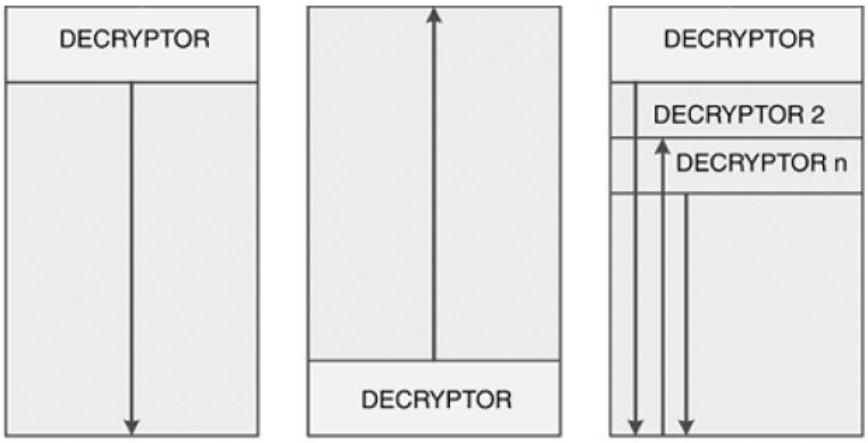


Figure 5: Use of multiple encryptions[1]

2.6.2 Polymorphism

Polymorphic viruses were discussed in Section 2.1.1. In this section we discuss polymorphism as a general obfuscation method instead of the virus-specific implementation.

Polymorphic malware can be viewed as the next step in the evolution of malware after encrypted malware [38]. The reason is that when encrypted malware was introduced the malware detection systems started targeting the decryption routine in order to detect encrypted malware. Hence signature-based detection remained just as effective. This chain of events lead to the evolution of polymorphic malware, which randomly change the decryption routine each time it infects a host [38]. Since the decryption routine changes for each infection it becomes challenging for signature-based systems to perform detection.

There exist counter mechanisms however that are able to detect such malware. For instance AV scanners that utilize sandboxing and emulation to run malware [38]. This is effective since the malware is run dynamically and decrypted, which allows for inspection of the actual malicious code. From this code the antivirus-vendors can create signatures for signature-based detection or observe malicious behavior for anomaly-based detection.

2.6.3 Metamorphism

Since polymorphic malware could be detected by emulating malware or running it in sandboxed environments the malware authors developed metamorphic malware able to change itself from infection to infection. Thus, like polymorphic malware the malware itself is changed upon infection. However, it is not simply the decryption routine which is changed, but the malware itself. This is a vital difference, as the malware need to edit code, then recompile [38]. As a result the complete malware changes over time and makes signature-based detection obsolete, as no parts of the malware can be identified by signatures.

2.6.4 Packers

Packers are commonly known as software applications that store either encrypted or compressed executables (packed), in such a way that when executed the packed executable is loaded into memory and executed [3]. Since packers both encrypt and compress they are also commonly called cryptors and compressors [41].

Due to the compression and encryption of packers it is typically challenging to analyze packers statically. For malware authors this is beneficial, as a lot of today's malware detection systems utilize static detection, for instance by scanning files stored on disk. This will be completely ineffective, as the malware only can be detected in memory. Hence it is resilient against most static analysis, except those that implement decryption and compression routines. However this can easily be avoided by implementing unknown or less commonly used encryption and compression algorithms.

Packers are typically categorized into the four following categories: compressors, crypters, protectors and bundlers [3]. These are defined below:

- Compressors: Simply compress executable, typically with little or no anti-unpacking tricks [3]. Examples of compressors include: Ultimate Packer for eXecutables (UPX) [42] and ASPack [43].
- Crypters: Packers that encrypt and obfuscate executables [3]. Examples include: Yoda's Crypter [44] and PolyCrypt PE [45].
- Protectors: A hybrid packer that utilizes both cryptor and compressor features [3]. Examples include Armadillo [46] and Themida [47].
- Bundlers: A type of packer that packs multiple executables and data files into a single executable [3]. Examples include PEBundle [48] and MoleBox [49].

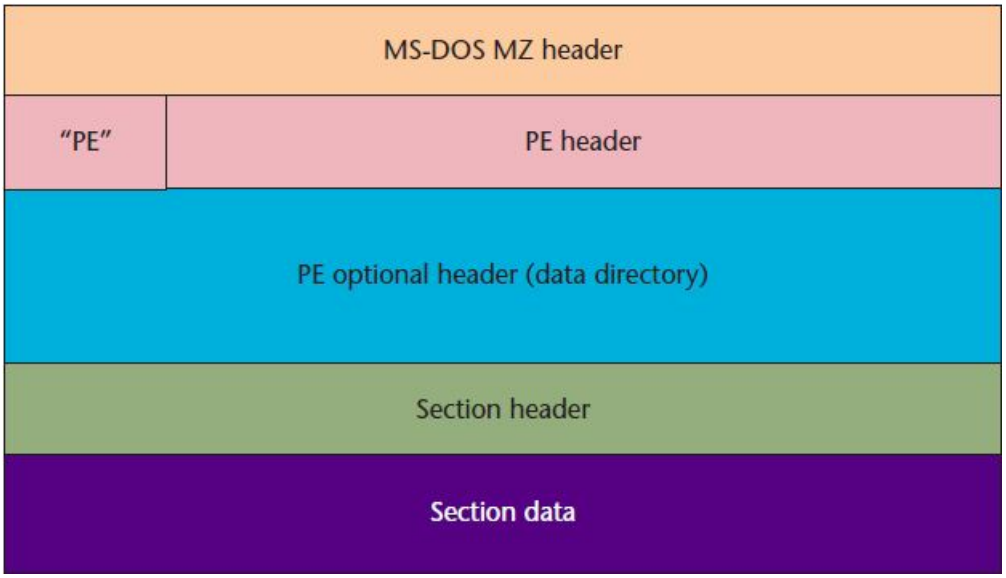


Figure 6: Portable Executable File Format[3]

Packers are most common for the Windows Portable Executable (PE) Format, however also exist for ELF (Linux executables). The PE file format is displayed in Figure 6. To perform packing a packer typically parse the PE internals structures, then reorganize PE headers, sections, import tables and export tables into new structures and then attach code that malware access before the original entry point(OEP) [3]. This code is commonly known as stub. The stub's function is to decompress or decrypt the original data and locate the OEP [3]. Packer software typically contains numerous different encryption and compression schemes, which allows for a range of different packed files.

Furthermore packers can utilize other forms of obfuscation such as polymorphism in order to further create new packed samples [3].

Crimeware as a Service

In 2009 a web-based automated packing service called Polypack was developed [50]. Although a research-based project, it provided people with the ability to test malware. More specifically, a front-end web-based interface to upload malware, which is packed with a set of packers, whose obfuscation resilience is tested against a set of antivirus software. Some would argue that this is in fact crimeware as a service [51], as one is able to test the effectiveness of packed malware before launching an attack. The project is not only able to find the packer with best average result, but also the best packer for each specific malware [50], to maximize the chances of not being detected. The project provides an example of how dangerous packer techniques can be to successfully avoid detection.

2.6.5 Specific obfuscation methodologies

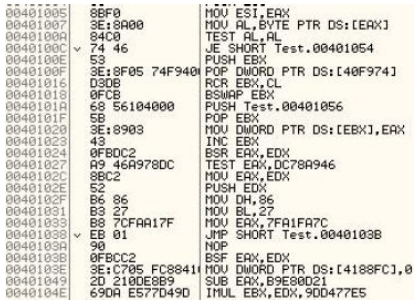
In this subsection we discuss a series of specific obfuscation techniques that are utilized either by themselves or together to obfuscate malware. Many of these techniques are used by polymorphism, metamorphism and packers.

Dead code insertion

As the name states, this obfuscation technique revolves around the idea of inserting dead code or useless code into the malware. This might for instance be no-operation pointers, changing state of program back and forth [38], or introducing code which is never executed. This method is effective against both signature-based detection and sequence-based detection. It is effective against signatures, as it insert code. Subsequently the effectiveness of the method depends on the size of the signatures. If the signatures are small and very specific, the method might fail, as it might not be able to insert code into that specific code segment. Thus, the larger the signatures, the more effective this method will be. Similarly, inserting dead-code into the malware will distort the sequence of execution. Hence it is likely to impact the sequence-based detection methods. However, its effectiveness depends on the implementation and method. Figure 7b shows a dead code injection example. The corresponding non-obfuscated sample code is provided in Figure 7a. As one can see from the example, these are simple NOP functions. These functions can easily be avoided by AV signatures, by ignoring such functions as a part of the signature [40]. This process is performed by a deobfuscator, whose purpose is to implement data-flow analysis algorithms to automatically remove irrelevant functions [7]. However, this in turn can be avoided, as there are many ways of doing nothing. As such, any programmer can include large portions of code which distort the signature, but have no significant impact on the execution and end-result.

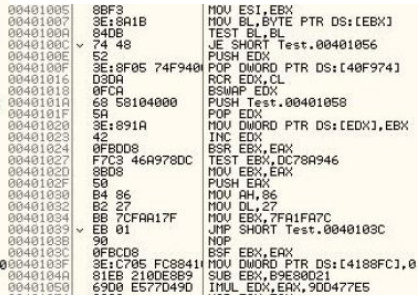
Register reassignment

Register reassignment refer to the change of registers used by live variables [38]. For instance during execution of a variable there exist a register R1 that remain unused. This register can replace the variable's current register R2 for the duration of execution [38]. The method is likely to be effective against signature-based detection, as it changes the code. However it is not likely to be effective against anomaly-based methods or reverse engineering [38]. A simple example of register reassignment is provided in Figure 7c. If

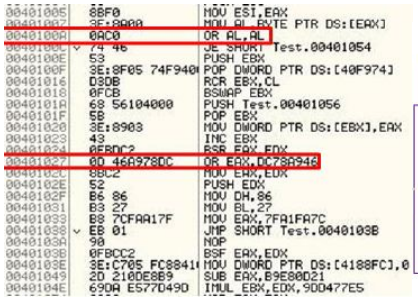


(a) Sample Code[40]

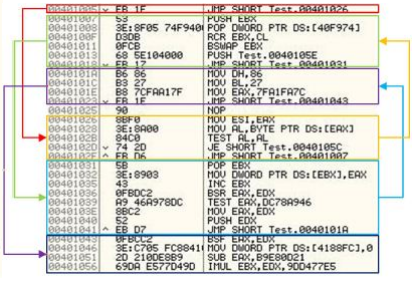
(b) Dead code injection[40]



(c) Register Reassignment[40]



(d) Instruction Substitution[40]



(e) Code Transposition[40]

Figure 7: Obfuscation Examples

one compare to the sample code in Figure 7a, one can see that the registers EAX, EBX and EDX are reassigned to EBX, EDX and EAX, respectively [40].

Code substitution

Code substitution, also called instruction substitution [38], is the process of substituting functions with equivalent functionality. This method is likely to be effective against signature-based detection, as it swaps out parts of the code. Previous research states that this is one of the most effective obfuscation mechanisms, which even might be effective against reverse engineering [38]. An example of code substitution is provided in Figure 7d. For instance, one can replace xor with sub and mov with push/pop [40].

Code-reordering

Code-reordering, also called code transposition [38]. The method is self-explanatory, as the name says, it is the process of moving code. This is typically independent code [38], as the result of execution should be the same. Similarly to dead-code insertion the code-

```
ASCII: var i = "foooo";
Hex: 7661 7220 6920 3d20 2266 6f6f 6f6f 223b 0a
ASCII: var  i     =     "foooo";
Hex: 7661 7209 2069 2009 2020 3d20 2020 2020 2022 666f 6f6f 6f22 3b
```

Figure 8: White space randomization. Source: [4]

reordering is likely to be effective against signature-based detection. This is because code is moved around, which should distort signatures and sequence. One limitation however, might be the number of changes one is able to implement. As the success of the method is likely to be dependent on the number of independent functions within the code. An example of code-reordering is provided in Figure 7e, where the independent functions are re-ordered to distort the signature.

Code separation

Separation of code and objects is a common obfuscation technique for many coding languages. For instance for PDF documents one can separate objects and code and store separately, such that they must be concatenated before execution [52].

White space randomization

White space randomization is a very simple technique of obfuscation, as it only introduces whitespace into the code. The goal is to make the code less readable or distort signature for signature-based IDS systems. This technique however can only be utilized on code that will be analyzed pre-execution, and not source code software that compile or assemble the software. Reason being that such compilers and assemblers ignore white space. For instance JavaScript parsers will ignore the whitespace [4]. However, since detection might be implemented as a step before the JavaScript execution, and not a part of the execution itself, the method is likely to be effective.

Comment randomization

Comment randomization is similar to white space randomization in the sense that it has no impact on the control flow of the program. Rather it introduces data that will be ignored by the parser or compiler. Its obfuscation impact is also similar to white space randomization, as it will make the code less readable and might trick signature-based systems. The last statement obviously fully depends on the method of the signature-based detection system. However, if comments aren't ignored, but implemented as a part of the signature, it will affect the obfuscation resilience.

String obfuscation

String obfuscation is a simple, but effective obfuscation technique for making strings harder to read. As an example one can split strings into several pieces, which are assembled before execution. Furthermore strings can be encoded using hexadecimal, Unicode and other encodings in order to create different representations of the string [4]. One example of such obfuscation is provided in Figure 9, where the string *we've got a problem* is represented by use of different encodings. Similarly to white-space and comment randomization, string obfuscation is ignored by compilers and parsers, such that it is only effective against analysis systems implemented pre-execution or reverse engin-

```
"we%27ve%20got%20a%20problem"

"%77%65%27%76%65%20%67%6F%74%20%61%20%70%72%6F%62%6C%65%6D"

"\x77\x65\x27\x76\x65\x20\x67\x6F" +
"\x74\x20\x61\x20\x70\x72\x6F\x62\x6C\x65\x6D"

"%u0077\u0065\x27%76%65%20\x67%6F%74\u0020%61%20%70%72%u006F%62\x6C%65\x
6D"
```

Figure 9: String obfuscation [4]

```
randomFunctionName = unescape;

function2 = eval;

var A1 =
randomFunctionName("%61%6c%65%72%74%28%22%77%65%27%72%65%20%67%6f%74%2
0%61%20%70%72%6f%62%6c%65%6d%20%58%65%72%65%22%29");

function2(A1);
```

Figure 10: Variable name randomization and function pointer [4]

eers.

Variable and function name randomization

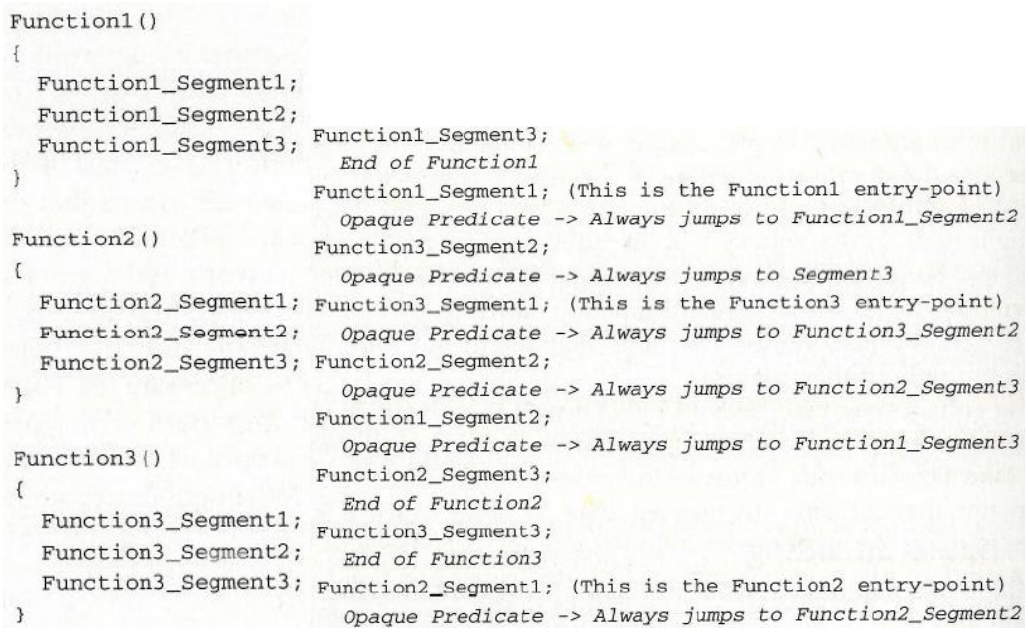
The variable and function name randomization takes advantage of the fact that functions and variables can be reassigned [4]. Hence it reassigns functions and values in order to confuse human analysts and detection systems. The technique can be viewed in Figure 10, where the variable A1 is set to randomFunctionName and used as a parameter in Function2, which is renamed to the eval function, that evaluates and executes the JavaScript. This technique is quite simple, but also effective as it immediately makes the code less readable. Fortunately it can be easily analyzed by for instance changing the function2=eval to function2=print, such that the code is parsed then printed to screen instead of executed. By using this method the encoding would be parsed as well, which drastically speed up the analysis.

Encoding

One simple way of obfuscation is to use different encodings. For instance encode parts of the program or variables differently. Although two of the most common encoding schemes are ASCII and Unicode, there exist a large set of encoding schemes. Furthermore one could implement new techniques or binary offsets to confuse the analysts. This method should be implemented at binary level, as some compiler may modify it during the compilation process [7].

Opaque predicates

The idea behind opaque predicates is to create logical statements whose outcome is constant and known in advance [7]. For instance creating statement that will never be true, can be used to confuse both human analysts and automatic decompilation tools [7]. For instance *'if (x+5 == x)'*, will never be true. For opaque predicates one seek to create such constants that are known in advance, but are hard to predict without running or debugging the code [7].



```
Function1()
{
  Function1_Segment1;
  Function1_Segment2;
  Function1_Segment3;
}

Function2()
{
  Function2_Segment1;
  Function2_Segment2;
  Function2_Segment3;
}

Function3()
{
  Function3_Segment1;
  Function3_Segment2;
  Function3_Segment3;
}
```

```
Function1_Segment3;
  End of Function1
Function1_Segment1; (This is the Function1 entry-point)
  Opaque Predicate -> Always jumps to Function1_Segment2
Function3_Segment2;
  Opaque Predicate -> Always jumps to Segment3
Function3_Segment1; (This is the Function3 entry-point)
  Opaque Predicate -> Always jumps to Function3_Segment2
Function2_Segment2;
  Opaque Predicate -> Always jumps to Function2_Segment3
Function1_Segment2;
  Opaque Predicate -> Always jumps to Function1_Segment3
Function2_Segment3;
  End of Function2
Function3_Segment3;
  End of Function3
Function2_Segment1; (This is the Function2 entry-point)
  Opaque Predicate -> Always jumps to Function2_Segment2
```

(a) Sample Code[7] (b) Interleaving obfuscation example[7]

Figure 11: Interleaving example

Table interpretation

Table interpretation is an obfuscation technique that converts the entire program or functions into a table interpretation layout [7]. The general idea is to break code into multiple short code fragments, while the code loop through a conditional code sequence that decide which of the code fragments to jump to [7]. The technique is known to be powerful, as it can confuse both human analysts and deobfuscators [7].

Inlining and Outlining

Inlining is a compiler optimization technique that replace a function call with the copy of the code for that code [7]. The goal is to improve runtime performance, by eliminating overhead of calling a function [7]. For obfuscation purposes inlining can be effective, as it eliminate the internal abstractions created by the developer [7].

Outlining was proposed in [39] as an obfuscation method, where one take selected or random code segments and create functions [7]. This will decrease readability and is likely to make manual analysis more challenging.

Code Interleaving

Interleaving is an obfuscation technique where the idea is to interleave two or more functions such that the code become challenging to interpret [7]. A sample interleaving obfuscation example is included in Figure 11. The obfuscated pseudo code in Figure 11b use opaque constants for jump functions to further obfuscate, as a simple jump instruction would be simple to follow [7].

Code Integration

A newer, more advanced, and far more potent form of obfuscation is called code integration. Code integration was introduced in the malware Zmist [51]. Where it integrated itself into the target program. First the malware decompiled executable into objects, then

it mutated itself using a combination of previously discussed obfuscation methods, before it finally inserted itself between the objects and reassembled the executable [51].

Anti-debugging techniques

As debugging is a common tool for reverse engineering it has become common for malware writers to implement tricks and techniques that detect debugging. Once detected the malware won't behave as it usually does, hence obstructing the analysis. Since debugging is supported by hardware and software there exists a range of different methods that can be utilized to detect debugging. Furthermore, since it is hardware supported, a lot of the techniques may be platform specific [1]. A vast range of techniques are proposed in [1]. It is outside the scope of this thesis to discuss all of these. However we will list them for completeness:

1. Hooking INT 1 and INT 3 on x86
2. Calculating in the interrupt vectors of INT 1 and INT 3
3. Calculating checksum of the code to detect break points
4. Checking the state of the stack during execution of code
5. Using INT 1 or INT 3 to execute another interrupt
6. Using INT 3 to enter kernel mode on Windows 9x
7. Using INT 0 to generate a divide-by-zero exception
8. Using INT 3 to generate an exception
9. Using Win32 with IsDebuggerPresent() API
10. Detecting a debugger via Registry Keys Look-Up
11. Detecting a debugger via driver-list or memory scanning
12. Decryption using the SP, ESP (Stack pointer)
13. Backward decryption of the virus body
14. Prefetch-queue attacks
15. Disabling the keyboard
16. Using exception handlers
17. Clearing the content of debug registers
18. Checking the content of video memory
19. Checking the content of the thread information block
20. Using the createfile() API
21. Using hamming code to attack break points
22. Obfuscating file formats and entry points

Anti-disassembly

Disassembly is the process of translating machine code into assembly language. Disassembly is a common tool for reverse engineering and even sometimes automated malware detection. Because of this, there has been developed methods that seek to disrupt

the disassembly process or distort the output, in order to avoid detection. Common methods for anti-disassembly are described in [1], and include the following:

- Encryption
- Polymorphism
- Metamorphism
- Code confusion to avoid analysis
- Opcode mixing-based code confusion
- Use of checksums
- Compression

2.6.6 Discussion

We have in this subsection discussed a range of different obfuscation techniques. What it is interesting is that code integration is one of the most advanced techniques. Yet this technique was first seen in the malware Zmist which was discovered in 2000 [51], which in a rapidly evolving digital world is actually quite some time ago. This in turn begs the question whether the evolution of obfuscation techniques has halted? Are the techniques so effective that no effort is needed to improve them? Yet the whitepaper from Symantec [52] state that new techniques are found every week. Hence it is likely that there exist novel techniques or combinations which are applied in the wild, but not covered here or even remain undocumented in academic literature.

3 Related work

This chapter contains the related literature of the thesis. First it discusses function call analysis and the differences of user mode and kernel mode detection. Then the state of the art with regard to malware detection for function calls is outlined. Finally an introduction to information-based dependencies is given, which describes how one can create dependencies between function calls.

3.1 Function call analysis

This thesis focus on detection by use of function calls, whether it is library calls in user mode, system calls in kernel mode or the hybrid function calls. Because of this, we will give an introduction to each layer in this section, and explain what we mean by the different layers.

3.1.1 The Operating system

The operating system can be viewed as a resource manager whose primary tasks is to keep track of which resources programs use, grant resource requests, account for usage and mediate conflicts [53]. When a program execute, the operating system start a process, which typically consist of multiple threads to carry out execution tasks. Library calls and system calls are collective terms for such tasks. Key difference being that library calls are execution tasks for user mode execution, while system calls for kernel mode execution.

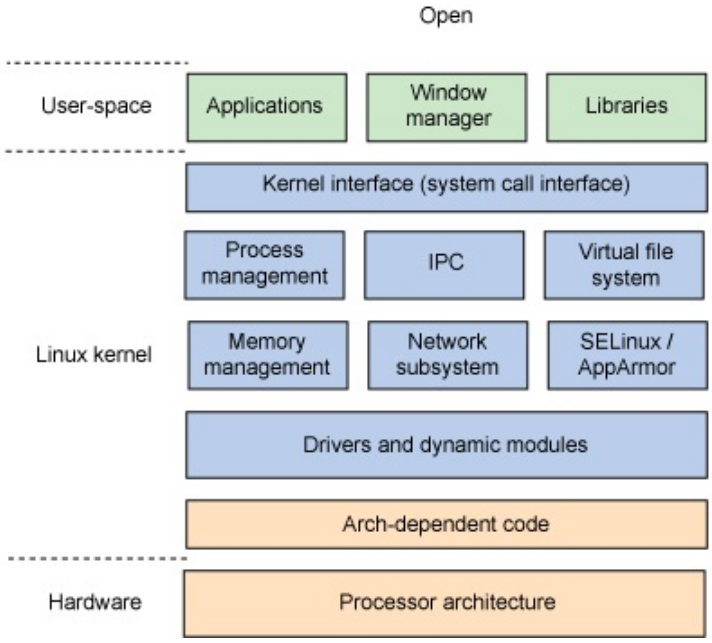


Figure 12: Linux Kernel Structure [5]

Kernel mode execution consists in general of file management, memory management,

process management, IPC (inter process communication), networking and SELinux, which is a feature for access control on Linux. Basically all of these system services are executed by system calls in kernel. An overview of the Linux kernel is provided in Figure 12. From this figure one can see how general applications and libraries run in user mode. Whenever a process needs access to a system service a trap is executed to transfer control to the operating system [53]. The operating system then inspect the trap function with parameters to figure out what is needed, then execute the appropriate system call and finally return control to the process [53].

To summarize a library call is execution of a function in user mode, system call is execution of function in kernel. While function calls is a collective term used for the hybrid execution that trace both user- and kernel mode execution.

3.1.2 Differences between library call and system call detection

Based on the fact that library calls are user mode execution while system calls handle system services it is reasonable to believe that there might be a difference in detection at the different layers. This assumption stem not only from the fact that system calls handle more critical system resources, but the fact that the amount of executions at the different layers might be different. Furthermore there are far less system calls than library calls. Subsequently the layers are quite different, with regard to access to critical resources, amount of calls and executed calls. This in turn might result in different detection rate, obfuscation resilience and throughput for the layers.

The reason that the amount of existing calls is different is that the hierarchy of libraries works as a funnel. High layer libraries utilize lower level libraries. As such, there exist a lot of high level libraries, while the lower one get, the fewer libraries there is. This idea is illustrated in Figure 13a. This figure is a Windows example, as the ntdll.dll is the interface to Windows kernel. However, the general structure is similar for both Linux and Windows.

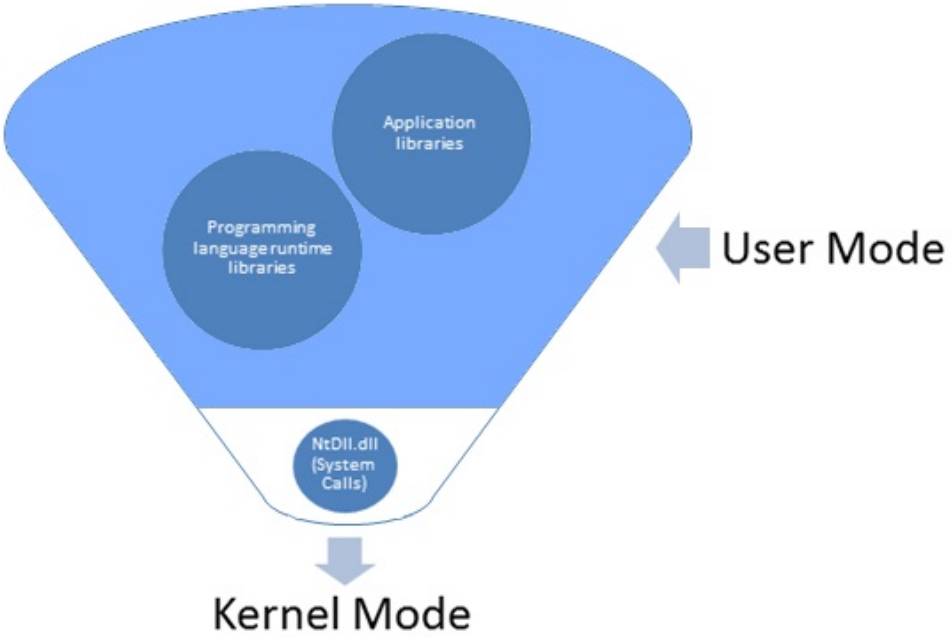
The hypothesis is that malware detection will be easier at a higher layer, since one see exactly which function calls that are responsible for the malicious behavior. On the contrary detection may be more challenging at lower layers, as the amount of traced calls is likely to increase. In addition there might be less visible malicious behavior. Thus it becomes more challenging to differentiate between malicious and benign. This is illustrated in Figure 13b. However detection at higher layer is likely to be susceptible to obfuscation as there exist enormous amounts of different library calls and not all are known. As mentioned there are substantially more library calls than system calls. However the library calls utilize the system calls. Thus it is reasonable to presume that library call analysis provide better throughput, as fewer calls are executed and traced.

3.2 State of the art

In this section we provide an overview of the state of the art for malware detection with regard to function calls. The section outlines different methods of performing detection with regard to the different analysis methods.

3.2.1 Overview

A non-exhausting listing of the methods and existing research is provided in Table 1. The table is categorized based on analysis method and detection model. Given the enormous focus on malware detection throughout the years there exist a lot more research. How-



(a) Function call funnel

(b) Actual Malicious Behavior

Figure 13: Library call compared to system call

ever surveying all material would be an enormous amount of work and a thesis by itself. Thus this project present what we believe is relevant with regard to the project and its research goals.

Control Flow Graphs Construct and traverse node with regard to node content and inter-relationship

Intrusion Prevention System A system whose task is to report and prevent any malicious activity

Tainting The process of marking data and monitoring and checking its flow throughout the system

N-gram Collection of n-byte strings which are typically used for detection based on statistical analysis or learning

Sequential Take sequence of function calls into account

Non-Sequential Disregard function call order. Simply counts occurrences

	Anomaly-based	**Signature-based**
Static	N-gram [54, 55, 56, 57], Tainting [58, 59, 60] Other [61, 62]	Control Flow Graph-based [63, 64, 65, 66, 67, 68, 8], Other [69, 70, 71]
Dynamic	Sequential [72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 11, 87, 88, 89, 90], Non-sequential [91, 92, 93, 94, 95, 96, 97]	Reference Monitor-based (Host IPS) [98, 99]

Table 1: Existing research on Function Call-based malware detection

3.2.2 Static signature-based detection

Previous work demonstrated that commercial signature-based systems (virus scanners) are vulnerable against metamorphism and polymorphism [100]. The reason is that these signature-based systems most often are syntactic and doesn't analyze the semantics of malware. To mitigate this issue semantic-aware malware detectors was developed [64, 65, 66, 67, 68, 8]. These semantic-aware methods are typically performed through static signature-based detection. The semantics is represented by use of control flow graphs (CFG), which describe the malicious code behavior. CFGs typically utilize source code analysis or disassembly to inspect behavior. This information is then used to construct and traverse nodes with regard to node content and inter-relationship.

As one can see in Table 1, most of the surveyed static signature-based methods are control flow graph-based (CFG). Control flow graphs were described previously. There exist however a range of other static signature-based detection methods. For instance detection based on disassembly and association rules [69], data-oriented behavior modeling based on source code analysis [70] and detection by use of statistics and similarity scores [71].

3.2.3 Dynamic signature-based detection

The dynamic signature-based methods are based on reference monitors and work as a host intrusion prevention system. A reference monitor is a validation mechanism whose goal is to enforce access control policy [101]. Subsequently an IPS based on this mechanism would be able to control execution access.

3.2.4 Static anomaly-based detection

The surveyed methods of static anomaly-based detection are mostly taint-based and n-gram-based. Tainting is the process of marking data and tracking its flow through the system. N-gram detection on the other hand divides data into n-grams and performs analysis based on these, whether based on learning or statistical analysis. Other static anomaly-based methods include methods that utilize disassembly with neural networks [61] and learning based on features from decompressor tools [62].

3.2.5 Downsides of static-based approaches

Recent work however prove that static analysis alone is insufficient, as obfuscation techniques can be applied to trick semantic aware detection [64, 41]. The paper [64] sought to exploit the fact that detection was performed statically, by including an opaque constant which only can be determined dynamically [64]. The code is then implemented in

such a way that certain behavior occur only given a certain condition/constant which is extracted during runtime [64].

Another technique which greatly decreases the effectiveness of static detection is the use of packers and cryptors. Packers and cryptors seek to compress and/or encipher the malicious sequences in such a way that they cannot be detected pre-execution. This of course depends on the detection module's ability to decompress/decipher, but in practice it should not be difficult to apply an algorithm which is unknown for the detection module.

It is obvious that both signature-based and static-based detection has challenges with regard to obfuscation techniques. To thwart these vulnerabilities focus shifted towards dynamic anomaly-based detection. The advantage of dynamic anomaly-based detection which utilizes function calls is that existing obfuscation techniques are very ineffective. Metamorphic and polymorphic code for instance, seeks to change the code in order to fool signature-based systems. But since the effect and behavior would be the same, function call-based detection should be more resilient. Furthermore it should be highly resilient against packers, cryptors and opaque constants since these are designed to trick static analysis. Subsequently it is reasonable to assume that dynamic anomaly-based detection based on function calls have greater obfuscation resilience.

3.2.6 Dynamic anomaly-based detection

For categorizing dynamic anomaly-based detection we chose to differentiate between sequential and non-sequential. I.e. whether the detection method take the sequence of function calls into account or not. This is an important difference as function calls might be malicious only in combination with others. However sequential methods have downsides as well, such as susceptibility to obfuscation [96]. For instance by introducing calls to distort the sequence. This is but one of many ways to categorize. There exist methods which focus on arguments, resource use, tainting, graph matching, statistics etc. However by categorizing based on sequence we could easily differentiate between all the surveyed methods.

Tainting is used for both static [58, 59, 60] and dynamic [87, 88, 89, 90] anomaly based detection. The static method use disassembly to analyze possible paths, then check code to detect all possible flows of information. While the dynamic method trace and control information. Tainting is sometimes called information flow control and typically used to determine accessibility of sensitive information or critical functions.

3.3 Information-based dependency matching

The realization that dynamic anomaly-based detection techniques are likely to have better obfuscation resilience lead to the creation of this thesis. Most of the function call-based methods surveyed in Section 3.2.6, used either sequential or non-sequential function call analysis. Furthermore some methods utilized the arguments and other dependencies between the function calls to perform detection.

MINIMAL[1] was one such method, that utilized sequence and arguments to create dependencies between function calls [11]. MINIMAL was a novel technique that proposed to use system call traces and their inter dependencies represented as malicious behavior graphs for detection. Since the system call traces don't contain dependencies by them-

[1] MINIMAL: *is a technique for mining minimally malicious behavior* [11].

selves these were created post-execution. This was performed by using the following three methods:

- Def-use dependence: *Express that a value output by one system call is used as input to another system call* [11]
- Ordering dependence: *states that the first system call must precede the second system call* [11]
- Value Dependence: *Logic formula expressing the conditions placed on the argument values of one or more system calls* [11]

The dependency rules described in MINIMAL form the base from which dependencies are used in this thesis. However, due to the fact that some of these methods might infer dependencies where there in fact are none, compel questions whether this method is reliable. This was to some extent discussed in [11], but further analyzed in [12], which also provided source code of def-use dependency matching. Examples include def-use dependencies based on integer values as described in [12]. This imply that a dependency exist, if a function call with argument x succeed another function call which use x as a return value. Now this is obviously not true, as use of integer is common in programming as both arguments and return values. However, for use of memory values it might be more reliable. Analysis of the reliability of dependency matching is of key importance in this thesis and analyzed further in Section 6.1.

There exist however other ways of inferring dependencies. For instance taint tracking which is proposed in [74]. The method proposed in [74] used an extended version of Anubis [102] to trace system calls and utilize taint analysis to create dependencies. Unfortunately taint-based systems result in high overhead [87], and will not be used in this thesis.

4 Graph Matching

This chapter provides an introduction to the graph matching problem. First it discusses what graphs are, then the different graph matching methods. The chapter finalize with a chapter on graph learning and SUBDUE, which is the tool used in this thesis.

4.1 Terminology

A graph consists of a set of vertices and edges. Vertices are also sometimes called nodes or points. A typical annotation of a graph is G=(V,E), where G is the graph, V is the vertices and E is the edges.

The size of the graph is the number of edges [6].

If two vertices (u,v) are connected by the edge e. The vertices are adjacent or neighbors [6]. This can also be denoted by e=(u,v). If there is no direction however, the edges are called undirected [6]. If all edges have directions it is said to be a directed graph [6].

Edges and vertices may also contain more information. For instance, both may contain a label. If a graph consist of edges and vertices with labels, it is called a labeled graph [6]. However, vertices and edges may also contain attributes. If this is the case, then the graph is called an attributed graph [6].

4.2 The graph matching problem

Graph matching is as the name suggests the problem of matching graphs. This may be performed by a various different methods. For instance one can use similarity measures for calculating how different the graphs are, or by use of other measures that represent the features of the graph. One of the most common methods of performing graph matching is by use of graph edit distance (GED). The graph edit distance can be defined as the minimum amount of edit operations required to transform one graph into another [103].

There exist two key methods of performing graph matching. These are exact graph matching and inexact graph matching. Exact graph matching seeks to find graphs that are identical, while inexact graph matching seek graphs that are as similar as possible. This can be performed by use of edit distance, such that for exact graph matching the edit distance of two graphs would be zero. While the edit distance for inexact graph matching should be based on some threshold. The taxonomy for graph matching is provided in Figure 14. As one can see there is both exact and inexact graph matching, which both have their subcategories.

In order to use graphs for malware detection one must find a way to build the graphs. Detection is based on a set of features, and the features for malware detection can be anything from amount of registry and file changes to the dynamics of function calls. If one take function calls as an example one can build graphs that represent the execution flow. These graphs can then form the baseline from which malware detection is performed. However it is not only full execution behavior that can be used. In order to improve detection rates, it is often common to prune graphs by removing either redundant or non-critical behavior. For instance by only using critical or security related calls for

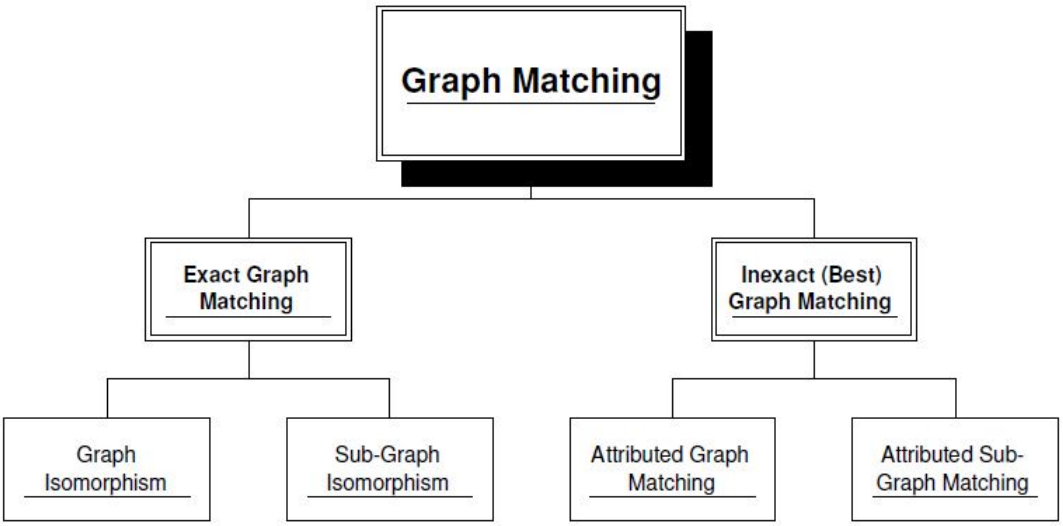


Figure 14: Graph Matching Taxonomy [6]

building the graphs. This might not only improve detection rate, but also throughput, as the graphs can become immensely large and complex.

The next subsections will go in further detail on exact- and inexact graph matching.

4.2.1 Exact Graph Matching

As mentioned above, exact graph matching is the challenge of finding identical graphs. That is, the graphs should have a one to one mapping. If this is the case, the graphs are said to be isomorphic [6]. As Figure 14 shows the subsections of exact graph matching are graph isomorphism and sub-graph isomorphism. I.e. whether the full graph is isomorphic or just a part of it.

For malware detection graph isomorphism is not desired. Reason for this statement is because programs change often, and that the actual execution behavior might depend on the environment specific variables, such as time and version of operating system. Furthermore, the execution context and options may change program behavior. Hence it is not unlikely that graphs may differ slightly based on which system, when and how it is executed.

Sub-graph isomorphism on the other hand is more resilient as one is able to create sub-graphs of malicious behavior. This was performed in [11] where minimal graphs for malicious behavior was extracted. This is not only beneficial in order to account for the vulnerabilities mentioned above, but is also likely to increase throughput, as the graphs are smaller. Another example is [104], where critical API graphs are deducted and used in a sub-graph isomorphic detection scheme.

The computational complexity of graph isomorphism remain an open question, since it has not been proven to have one type of complexity such as P or NP-complete [6]. Sub-graph isomorphism however, has been proven to be NP-complete [105]. The problem with NP-complete complexity is that the complexity rapidly increases with the size of the problem. The impact is that some graphs might be too large for classification. This will severely impact the method, as the throughput is directly associated with the size of the graph. Subsequently pre-processing or pruning graphs might prove a necessity.

4.2.2 Inexact Graph Matching

Inexact graph matching should be utilized wherever it is impossible to find isomorphism between the graphs [6]. This is typically the case in graphs which vary a lot or has high variance in terms of vertices, edges, labels and attributes. For instance, if function calls where represented with all parameters and return values, without pruning or pre-processing, then isomorphism would be very unlikely. This is because each call trace usually consists of a lot of environment specific variables such as memory values that change for each execution. Hence, if one look purely on the memory values and compares these for different traces, one is not able to extract useful information. However, the patterns of these environment specific variables, or other parameters might be important. For these instances, where isomorphism might not always possible, inexact matching can be used.

Inexact graph matching has been successfully applied in cartography, character recognition and medicine [6]. It is not as prevalent in malware detection, however there exist examples. For instance [106], where subgraphs are used for detection using graph edit distance. The method utilizes API calls, which is equivalent to function calls, I.E. both user mode and kernel mode detection.

The computational complexity of inexact graph matching is NP complete where:

$$|V_M| \leq |V_D|$$

[107]. Inexact sub-graph matching is also known to have NP-complete [6]. This has the same ramifications as explained above.

4.2.3 Graph-based Learning

In this section we introduce graph-based learning. The first section describes what machine learning is about. While the second section discusses machine learning with regard to graphs.

Machine Learning

Machine Learning is defined as the scientific discipline whose goal is the classification of objects into a number of categories or classes [108]. This allows for an automatic method that is able to classify and differentiate data into classes, based on learning from the input data. For graphs this is the automatic classification of structural patterns such as vertices and edges, and non-structural data such as vertice- and edge attributes.

Graph-based learning

According to [109], graph-based learning can be categorized into multi-relational data mining and learning, and graph-based relational learning. The latter is implemented in a system called SUBDUE [110], which is useful for graph-based discovery of patterns, clustering and supervised learning.

For this thesis graph-based relational learning was chosen, based on the fact that it is fully documented, automated, and allows for graph-based learning. One key downside however in SUBDUE is that it only looks for structural patterns based on vertices, edges and labels. This means that one cannot perform attribute-based learning.

SUBDUE

SUBDUE is a graph-based knowledge discovery system that finds structural and relational patterns in data representing entities and relationships [110]. In SUBDUE graphs

are represented by labeled vertices and edges. The goal of SUBDUE is to find patterns in graphs, and more specifically it can perform unsupervised learning, supervised learning and graph grammar learning [110]. SUBDUE has been successfully applied in areas such as bioinformatics, web structure mining, counter-terrorism, social network analysis, aviation and geology [110]. SUBDUE has also been tested for anomaly detection on network data [111]. The method utilized weighted graphs, unsupervised learning for detection and 1999 KDD Cup data set. It found that both conditional substructure entropy and graph regularity could be used to find anomalous data in network traffic [111].

In order to learn and find patterns in data, SUBDUE seeks to compress subgraphs in order to find structural properties in data. There are several ways of measuring which subgraphs that are used. However one method is based on how much the subgraph compresses the total graph. Which means that SUBDUE will go through all subgraphs and try to compress. It then chose the subgraph that compress the most and this graph is used as a feature. The process of choosing and compressing graphs is displayed in Figure 15.

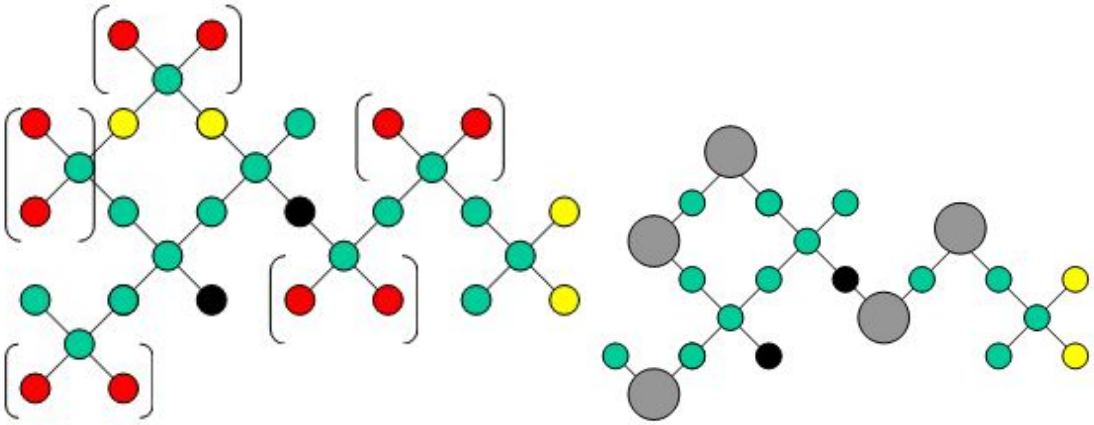
SUBDUE can be used for both supervised and unsupervised learning, as well as exact and inexact graph matching. For inexact graph matching a threshold can be set using the -threshold option. The supervised classification is performed by finding subgraphs that exist in one class, but less frequent in the other. Thus a set of subgraphs is checked to see which has the best relevance with regard to detection rate. Furthermore cross validation can be automatically performed by use of the cvtest program and -nfolds option.

SUBDUE has three evaluation methods for finding substructures available. These are minimum description length, size and set cover.

Minimum description length for substructure S in the graph G is calculated as following: value(S,G)=(DL(G) / (DL(S)+DL(G|S)). Where DL is the description length in bits, and (G|S) is G compressed with S [112]. MDL is an evaluation method that is good at compressing graphs.

Size for substructure S in graph G is calculated as: value(S,G)= size(G) / (size(S) + (size(G|S)) where size(G) = (vertices(G)+edges(G)) and (G|S) is G compressed with S. Size is a bit faster to compute than MDL, but less consistent [112].

Set cover of substructure S is computed as the number of positive examples containing S, plus the number of negative examples not containing S, all divided by the total number of examples [112]. Set cover is considered the method which is best at discriminating between classes.



(a) With Patterns (b) Compressed Patterns

Figure 15: SUBDUE Patterns

5 Choice of methods

This section discusses the methodology used to guide and analyze the individual research questions and the project in its entirety. It will discuss the different methods and justification for why the method is appropriate, as well as expected results. First the dataset is explained, along with evaluation of previous datasets and generation of the dataset used in this thesis. Then the pre-processing, where it is explained how the dataset is preprocessed before it is used for detection. Methods for detection classifiers are then analyzed in order to choose the most proper detection method with regard to the dataset and its features. Finally experimental design for each of the research questions are discussed where the methodology for experiments for each research question is explained and justified.

5.1 Dataset

The most important issue when dealing with IDS systems and malware detection is a representative data set. The dataset should be a best possible representation of the actual environment. This means the base rate (attack frequency), attack types and data in general should be similar to that of the real environment.

5.1.1 Evaluation of existing datasets

There exist a few function call datasets which are frequently used for research [113, 114, 115, 116]. The first three [113, 114, 115] are the Defense Advanced Research Projects Agency (DARPA) datasets from Massachusetts Institute of Technology (MIT) Lincoln Lab. These are often referred to as DARPA or IDEVAL dataset, short for intrusion detection evaluation dataset. There exist three such sets, one from 1998, 1999 and 2000. All three datasets are simulated, which means that there are no background anomalies. The downside of these datasets is that they are very old and not representative for today's threats. Further deficiencies are discussed in [91], which state that too few calls exist for software traces in training data, making the training process unrealistically simple [91]. It also list the number of system calls is limited [91]. Furthermore the attacks are listed in a separate list with timestamps. Such that one has to extract the function call traces based on timing of attack. Additionally data is stored in the Basic Security Module (.bsm) format. Subsequently, some effort is required to make use of this dataset. Because of this, the fact that it is outdated and the downsides discussed in [91] it will not be used for this thesis.

The Sequence TIme-Delay Embedding (STIDE) [116] dataset on the other hand is well organized and easy to understand. It has separate traces for training and testing data for both malicious and benign data. The key downside with this dataset however is that it only contains system call traces and not library calls. Furthermore it only traces system call names and not parameters and return values which is essential for this thesis. As a result we have no suitable dataset, and have to generate our own.

5.1.2 Generation of dataset

The advantage of generating a new dataset is that one can use new types of malware, which make the dataset more relevant. Furthermore one can carry out traces for system calls and library calls. Resulting in a dataset which can be used to research differences in detection of the different layers.

To generate a dataset, the following is needed: a secure environment for tracing, benign software, malicious software and tracing/hooking software.

Secure Environment

There are several ways of setting up an environment for testing. Examples include virtualization, emulation and hardware. These are discussed below.

Virtualization

A secure environment for tracing and testing malware can be achieved by setting up a virtualized environment using VMware [117] or VirtualBox [118]. This is common practice by reverse engineers and there exist several guidelines for how to do this securely [119]. The advantages of using virtualization are that it is in general quite fast as it can be scripted. Furthermore it is considered safe, as one can isolate the environment. There exist however virtualization aware malware that, if advanced and successful, might be able to escape the virtualization. These, and more security considerations of virtualizations are discussed in Section 2.5.2.

Another potential downside of this environment is the fact that it is isolated. A lot of malware and software alike need external resources and commands to operate. For instance botnet malware might not be active until it receives commands from a botmaster. Similarly a worm might not propagate if it cannot see active network interfaces. To counter this it is possible to set up a network interface on the virtualized machine, then install FakeDNS to emulate DNS responses [120]. Thus if a malware seek to capture information then send a mail, this information will be traced. Of course, the e-mail and any other network traffic will be sent to a 127.0.0.1. So no response will ever be available. This means that malware that needs interaction with a botmaster or similar, will not be able to show its true behavior. This is a necessary limitation as it would not be ethical to let a machine become infected and stay connected to the Internet. The ramification of such an event might be responsibility for infecting other hosts, or being used as a bot or proxy during an actual attack.

Emulation

An alternative to virtualization is to use emulation services such as QEMU [121]. QEMU was used as a platform in [74], where they state that the issues of virtualization aware malware is not applicable as QEMU is not virtualization, but emulation. This is true, but there are other issues need to be considered when using emulation. The most severe issue is that emulation is far from a native environment [32]. Thus one cannot expect the software or malware to behave as it originally would. Which is a serious issue, when tracing and analyzing the behavior of malware.

Despite the issues above we tested Zerowine and Zerowine Tryouts, which are malware analysis images that runs on QEMU. Zero Wine [122] and Zero Wine Tryouts [123] provided a lot of useful output, such as complete report of imported and used libraries, strings, file headers and signature. However, in both cases the API tracing crashed far too

often. It was in fact not able to provide API-trace output for any of the 10 malware that was tested.

Hardware

The third method of dynamic malware analysis is by utilizing physical hosts as environment. To do this one need a physical computer, with installed operating system and software. The malware must then be traced and extracted, before the physical computer is re-imaged. This is in general a slow process as one would have to re-image the computer after each infection. The upside of the method is that it is not affected by virtualization-aware malware.

Due to virtualization's scripting and snapshots, malware analysis can be both fast and scalable. Because of this, virtualization was chosen as a method for generating the dataset. However, virtualization was not used with emulated DNS responses due to t

Tracing software

There exist several software for both library call and system call tracing, however many of these tools use different techniques for tracing. Each of these techniques has different benefits and downsides that might affect the outcome of the traces. Subsequently it is important to understand these techniques in order to choose the tool which is the most accurate and complete in the sense that it is able to provide traces for all calls, regardless of library. An analysis of the different techniques is provided in [124, 125]. The techniques described in these resources are discussed with regard to their advantages and downsides for this project below.

Clone DLL

This method copies a DLL's functionality into a wrapper DLL and replace the original DLL file, such that the copied DLL is loaded by the executable [124]. Since one control the DLL one can easily implement functionality to trace all calls to the DLL itself.

- Pro: Easy to implement, complete control over calls and parameters [124]
- Con: Need to edit all DLL's one seeks to trace, as well as updating these over time [124]

Import Address Table Patching

Executable files on Windows are stored in the Portable Executable (PE) format [124]. This format contains an import address table, which store information about the imported functions from DLLs. The table contain entries for each DLL that point to a new table. This table lists all the functions of each DLL. This function-specific entry is updated every time that function is executed [124]. Since all of this is maintained in memory during execution it can be traced. The way this is performed is to replace the function-specific entries with hook function that cause all executions to be traced through the hook [124]. Subsequently one can trace both function call name, parameters and return value.

- Pro: IAT Patching is standard in Windows [124]. Furthermore one is able to trace all calls regardless of DLL
- Con: Complex implementation [124]

Minimalist Debugger

The implementation of a spying application that insert x86 "*'INT 3*"' as the first instruction for all APIs one wish to trace [124]. When these functions are executed a status breakpoint occur, and the spying application can extract API call information and call stack.

- Pro: Easier to implement than IAT patching [124]. More maintainable as status breakpoint API can handle all function calls [124]
- Con: Windows exception handling is slow, which cause scalability issues [124]. Furthermore it is difficult to catch the return value [124].

Remote Debugging using WinDbg

Remote debugging can be performed by setting up two physical computers where the client executes software with kernel debugging enabled. The server is then connected remotely via a serial null-model cable or high-speed Firewire [7]. Kernel debugging can be enabled by editing the boot.ini file on Windows. The output can then be parsed using WinDbg on the server [7].

- Pro: Both kernel-mode (system call) and user-mode (library calls) can be traced
- Con: Requires a complex physical lab-environment [126]

Ptrace for Linux

Ptrace for Linux is a system call that can be used by processes to observe and control the execution of another process. This is a Linux-based method which is implemented in Kernel. Meaning that it easily can be utilized without hooking into or kernel. Ptrace is utilized by both strace and ltrace, which are Linux-based software that can trace kernel mode and user mode execution.

- Pro: Both kernel-mode and user-mode can be traced
- Con: Linux only

Chosen methods and tools

In this thesis both Windows and Linux datasets were tested. Unfortunately the Windows data proved challenging to analyze as it was very unfamiliar with regard to structure and execution compared to Linux. Seeing as the dataset need to be analyzed and explained we choose Linux as a platform to create our dataset. The reason for this choice was mostly based on familiarity with Linux, as the trace files are quite complex, and we had limited time on this thesis. Because of this, it was easy to chose hooking technique as well, since Linux has implemented ptrace for this purpose as a part of its kernel. As a result both strace and ltrace are used for tracing system calls and library calls for Linux.

5.1.3 Dataset Generation Architecture

We have so far discussed all the aspects of setting up the environment for malware and software analysis. This subsection outlines the dataset generation architecture in its entirety. That is, how the dataset generation is implemented in this thesis.

In order to provide a safe an automated environment we chose virtualization. Virtualization can be scripted by use of the vmrun executable. This allows for virtualized machines to start, stop, create snapshots, revert snapshot, load files, extract files and execute files. First off, we created a virtual Ubuntu machine using Ubuntu 8.10 (Intrepid). The old version of Ubuntu was chosen due to its likelihood of being vulnerable, which in turn would enable the malware. This virtual client was set up without networking in order to contain the malware. The tracing software strace and ltrace was a part of the distro, so there was no need to install or change the system. A snapshot was then created to provide better control of the environment, such that it could be reverted after infection.

A script was then created on the host computer, which ran Windows 7 with VMware workstation. The script worked in the following way:

- Run additional script for all files in folder X that execute the following commands:
  Revert snapshot
  Start virtual machine
  Load software/malware from host to virtualized client
  Trace and execute software/malware and log to file
  Copy execution trace log from virtualized client to host
  Stop virtual machine

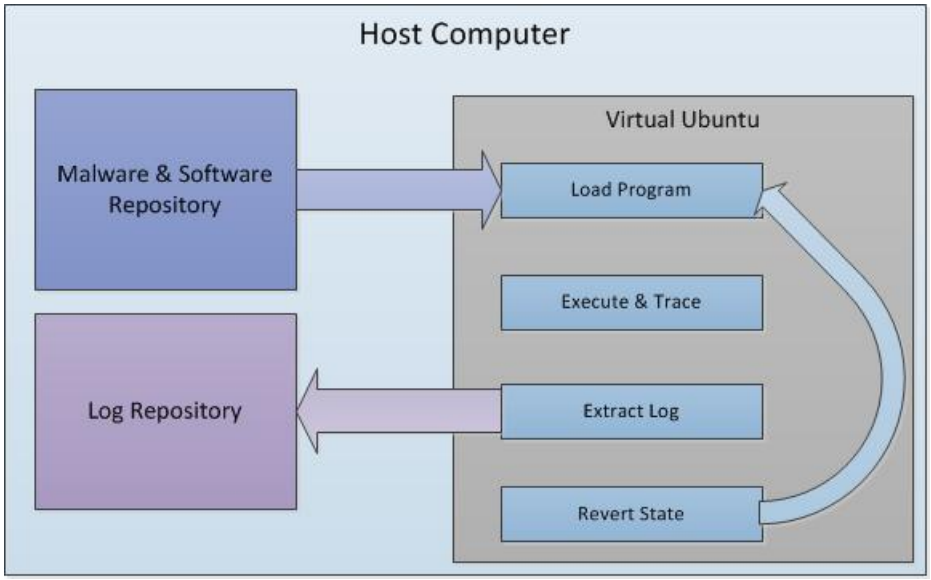


Figure 16: Dataset Generation Architecture

The process can be viewed in Figure 16. This is not only a simple process, but also secure and scalable. Seeing that an arbitrary amount of malware and software can be executed in a safe environment and logged to file by the push of one button. The process is performed three times for all software and malware, such that library calls, system

calls and function calls are logged separately. This has the inherent benefit that previous execution never will affect the next. The scripts that automate this process is provided in Appendix B.

5.1.4 Dataset selection

Software and malware can be downloaded from open sources on the Internet. Software can be downloaded directly from client through for instance Ubuntu Software Center. Malware on the other hand can be gained through sites such as VX heavens [127], Packet storm security [128] and Offensive computing [129]. For this thesis we will use VX heavens as they have a database of over 271.000 thousand malware samples and conveniently allow download through use of torrent (peer-to-peer file sharing).

Since we chose Linux as a platform, there is a limited amount of malware. Furthermore the malware is downloaded from online resources without description of capabilities. Subsequently it is challenging to analyze the malware' obfuscation abilities, as this would require that one reverse engineer every sample. This is obviously not feasible within the time limit of this thesis. Hence malware is chosen randomly based architecture and thus likelihood of success.

For this thesis 200 malware and 100 software was chosen and traced. There were performed three traces for each, which resulted in a total of 900 traces. Due to high complexity of graph matching, only trace-files with less than 150KB were chosen. Subsequently a subset of 190 malware and 75 software was selected as the final dataset.

In malware detection it is beneficial to have an equal amount of malware and software, such that they have equal statistical impact on the classifier. This can be mitigated by use of cross-validation. Cross validation make sure that the same amount of samples for each class is used to train the classifier, regardless of the classes' total amount of samples. For instance if one had 200 malware and 100 software, one can utilize 10-fold cross validation, which pick 10 software and 10 malware which is used for training of the classifier, the rest is used for testing the accuracy. This is then performed for all samples, such that both classes always have equal statistical impact on the classifier.

5.1.5 Dataset statistics

This section include descriptive statistics of the dataset that can be used to further understand the data used. Understanding the data is important, as it might reveal interesting patterns concerning the amount of vertices and edges for the different layers. Furthermore it gives an impression of the variance within the dataset. Table 2 include descriptive statistics for edges and vertices for all three layers of detection, for malware, software and both. The table includes total number of samples within each class, as well as sum, mean, max, min, median and mode for all vertices and edges. This can be used to compare the different layers of detection, as well as comparing software traces to malware.

Malware and software comparison

By comparing the total number of vertices for malware and software, one can easily see that there is a vast difference of vertices and edges. Software contains almost the double the amount of vertices and edges. This can be explained by the fact that malware generally have small tasks. They usually contain an exploit and a short payload, depending on the malware's purpose. Some malware may only be exploits, which seek to take advantage of a vulnerability, gain access, then open a reversed shell or install further malware.

	Software		Malware		Malware & Software	
Total	75		190		265	
	Vertice	Edge	Vertice	Edge	Vertice	Edge
Library						
Sum	8110	11146	4700	6178	12810	17324
Mean	108,13	148,61	24,74	32,52	48,34	65,38
Max	1343	2115	985	986	1343	2115
Min	0	0	1	0	0	0
Median	15	24	7	8	8	10
Mode	15	8	5	8	5	8
System						
Sum	9025	9536	7150	8001	16175	17537
Mean	120,33	127,15	37,63	42,11	61,04	66,18
Max	959	1000	315	438	959	1000
Min	24	26	22	24	22	24
Median	118	121	26	28	29	31
Mode	118	121	26	28	26	28
Function						
Sum	17183	22562	12496	17432	29679	39994
Mean	229,11	300,83	65,77	91,75	111,99	150,92
Max	1509	2244	1011	1730	1509	2244
Min	27	32	26	32	26	32
Median	132	162	33	43	39	48
Mode	120	142	30	38	31	38

Table 2: Descriptive statistics

While other malware might be more elaborate, for instance rootkits, that seek to subvert the system in order to stay hidden. Software on the other hand generally has more extensive tasks. They also often come with a GUI, which requires more processing. A list of the selected dataset for software and malware can be found in Appendix C. If one look closer at the Appendix, one see that there are 31 backdoors, 20 DoS applications, 62 Exploits, 30 flooders, 12 hackTools, 9 rootkits and 26 viruses. The software set is based on Ubuntu distro binaries, and not sorted by class or application area.

Detection layers

If one look at the different layers, that is user mode (library calls), kernel mode (system calls) and hybrid (function calls), one see that there is a difference in amount of calls. In contrast to kernel mode, user mode for instance does not include memory initialization and calls for exception handling. Hence it is expected to see less executions and hence less vertices and edges for user mode. This is reflected in the dataset and can be seen in Table 2. Furthermore, function calls represent the hybrid execution of both user mode and kernel mode. Hence, this layer can generally be viewed as the sum of library calls and system calls, at least in theory. In practice however there is a slight difference, since they might be represented differently by use of the tracing programs ltrace and strace. Strace is used to trace system calls while ltrace is used to trace library calls and hybrid calls. Furthermore, each trace is performed separately, thus each layer represent a separate execution. As such programs may execute differently based on scheduled runtime by the kernel and environment specific conditions. These conditions may vary, but might

for instance be time of day and other processes, since malware might look for certain conditions before it execute payload. The variance in these conditions however should be minimized as the execution is automated in the same clean environment for the whole dataset.

5.2 Pre-processing

This thesis is based upon the idea of combining behavioral detection and graph theory. In order to combine these we need to pre-process the data to form structural properties that can be used for malware detection. This is where information-based dependency matching is utilized. This was discussed in Section 3.3, where dependencies between vertices are created based on sequence or similarity of parameter or return values. To achieve this, a trace parser is needed that transform attribute data into structural data. This was performed in [12], where the source code is provided. The same source code was used in this thesis and extended to be able to parse function calls, system calls and library calls, as well as output to a format which is parsable by the detection engine. The source code is appended in Appendix A. The methodology concerning the reliability testing of information-based dependencies is further explained in Section 5.4.1.

5.3 Detection classifier

Previously we have discussed the tools and environment to create the dataset. This section focus on which classification method that will be utilized. That is, which method that is used for determining whether the data is malware or software. The goal is to choose a classification method that is able to utilize as many of the features as possible from the dataset. Furthermore it should be able to analyze structural properties such as graphs. The features that exist for the dataset is a set of traces for both malware and software. Each trace consist of a set of function calls, that have a name, return value, set of parameters and dependencies to other function calls.

5.3.1 Machine Learning and Pattern Recognition

Machine learning and pattern recognition was introduced in Section 4.2.3. Machine learning was the process of purposely changing behavior in order to increase performance [130]. While pattern recognition is the scientific discipline whose goal is the classification of objects into a number of categories or classes [108]. There exist several different methods within machine learning and pattern recognition. Broadly categorized these fall into either unsupervised classification and supervised classification. Unsupervised classification deal with classification of data based on similarity measures. While supervised classification is classification based on pre-defined features.

Both supervised classification and unsupervised classification can be implemented by use of programming or automatic tools such as Matlab or Weka [131]. This thesis we will focus on the supervised classification. The following subsections we will discuss the different machine learning classifier methods.

Statistical Models

One approach to machine learning is based on statistical modeling of data [132]. Statistical modeling is the application of probability theory and decision theory to get an algorithm [132]. Within statistical models one can classify between generative and discriminative models. A generative model is a probabilistic model of all variables. While a

discriminative model only model the target variables conditional on the observed variables. For instance, logic regression directly estimates parameters of P(Y|X), while naive bayes directly estimates parameters for P(Y) and P(X|Y) [132]. Generative models require more work, but can utilize more prior knowledge, needs less data, is more modular and can handle missing or corrupted data [132]. Discriminative on the other hand focus only on discriminating one class from the other. Can be more efficient once trained and fewer modeling assumptions are required [132]. Generative models include Naive Bayes, Mixture of multinominals, Mixture of Gaussians, Hidden Markov Models, Bayesian networks and Markov random fields. While discriminative models include logistic regression, Support Vector Machines, Traditional neural networks, nearest neighbor and conditional random fields.

A downside of the statistical models is that they work best with numeric values. Subsequently nominal values such as string values are often converted into a vector in order to get a numeric value [130], before they are used for classification. Unfortunately most of the values in our dataset are string values (nominal). This has the inherent downside that we would have to convert almost all parameters and return values.

Graph Edit Distance

Graph edit distance (GED) was mentioned in Section 4.2 and defined as the minimum amount of edit operations required to transform one graph into another [103]. Graph edit distance can be used for both exact and inexact graph matching. This is performed by looking at the GED, whether it is zero, or within a certain range.

Using GED for detection however, means implementing a signature-based system, as it is challenging to create a single GED that contain different malicious behavior. For instance a GED can be created for an execution graph of a specific malware or malware family. This is possible, since any similar malware or malware of that family is likely to have a similar execution flow. A completely different malware however is likely to have a very different execution flow, and hence a very large GED value. Because of this, GED-based detection, would result in a signature-based detection system, where for instance each malware family has a GED and detection is based upon some threshold.

Graph-based Learning

Graph-based learning and pattern recognition was explained in Section 4.2.3. This concept is similar to regular machine learning and pattern recognition, however it has a unique ability to analyze and find structural patterns in data. SUBDUE was mentioned as a graph-based learning method. Unfortunately SUBDUE has the key downside that it is only able to take vertice and edge labels into account, and not vertice and edge attributes [110].

5.3.2 Chosen detection method

One of the key goals of this thesis was to create an anomaly-based detection method. Since using GED would infer a signature-based method we ruled this out first. Both the implementation of regular- and graph-based machine learning and pattern recognition was tested. Unfortunately the downside of Weka, by not being able to use nominal values proved to challenging. Furthermore the thesis is not only based on learning and pattern recognition, but also graphs and structural patterns. Because of this we found it most suitable to use graph-based learning.

5.4 Experimental Design

This section discusses how this thesis will answer each of the research questions.

5.4.1 Reliability of information-based dependency matching

To test reliability one has to check whether false dependencies are inferred. This is challenging without fully knowing the software. Furthermore the complexity drastically increases when size of software increase. As a result it is almost impossible to test reliability using third party software. One way to perform this is to do synthesize testing. I.E. create a set of software programmed to test different scenarios. This was to some extent tested in [12]. This thesis will further examine the reliability issues, by performing synthesize tests for def-use dependencies, ordering dependencies and value dependencies.

Synthesize testing is performed by creating a sample program for specific scenarios until each type of variable is covered. That is, synthetic programs are created for different scenarios such as object handling, file IO, if/else checking, strings, etc. until all type of variables has been tested. All types of variables are considered good until a false dependency is found. If false dependency is found, the variable type is excluded from the dependency parser. The goal is to rule out all types of variables that might infer false dependencies, such that the dependency parser is able to infer correct dependencies. The reason for this goal is to tune the dependency parser in such a way that it only creates good dependencies, which result in correct execution behavior graphs.

Metrics

Metrics are commonly used to compare measurements to a predetermined baseline [133]. Measurements are objective observations of raw data [133]. Metrics on the other hand, are either objective or subjective human interpretations of those data [133]. Good metrics should be measurable, attainable, repeatable and time-dependent [133]. In order to create good metrics for our experiment we seek to define metrics which reflect these properties. Since the goal of this experiment is to find reliable variable types, the resulting metric is to find whether the variable type can infer false dependencies. This is measured by use of synthesize testing to find examples where a false dependency can be inferred. If no such examples are found the variable type is considered reliable for use of dependency matching.

Results

The results of this testing will directly impact the next section as it will limit the available features. More specifically, by limiting use of information-based dependency matching there will be less dependencies and subsequently less features. Thus it is of key importance to understand and make information-based dependency matching reliable, before continuing with other experiments.

5.4.2 Obfuscation resilience of information-based dependency matching

The goal of this research question is to find whether information-based dependency matching is resilient to the taxonomy of obfuscation methods presented in Section 2.6. To achieve this a set of synthetic programs are implemented. First a simple hello world is created to represent the original sample. Another program is then created for each of the obfuscation methods. The goal is to see whether changes in code is reflected in trace output and hence execution behavioral graphs which are used for detection. The obfuscated applications should be as simple as possible, but always include code which

"“enable"” or "“represent"” the obfuscation technique.

Metric

There exists previous work on obfuscation techniques [38], which proposed the following metrics: potency, resilience, stealth and cost. Where potency is defined as the amount of obscurity added to the program, resilience is ability to break automatic deobfuscator, stealth is how well it blends with program and cost is the added overhead [38]. In our analysis we seek not to analyze the methods full capabilities, but rather their direct impact on our detection method. As such we will not discuss all these issues, but rather focus on whether the obfuscation technique can be detected. That is, if the obfuscation method impact the trace output. The metric of this experiment is thus the impact of the obfuscation method. This can easily be measured as it is an objective observation of differentiation in execution flow.

Limitations

Due to limited time and complexity of certain methods we will be unable to test all the methods described in Section 2.6. Because of this we limit our experiments to packers and certain specific obfuscation methods. The reason we don't implement polymorphism and metamorphism, is because these utilize a set of specific obfuscation techniques. Thus by testing the specific obfuscation techniques, we will more accurately determine the impact. The specific obfuscation techniques that are not tested in our experiments are the following: encoding, code separation, table interpretation, code interleaving and code integration.

It is important to note that only a single experiment is implemented for each obfuscation technique and that no definite conclusions can be drawn from these experiments. The experiments merely provide a pointer to what might be the case, which in turn should be subject to further research.

Results

The results of this test are not necessarily complex to analyze, but to understand. Reason for this statement is that an obfuscation method that affect the execution behavior, doesn't necessarily mean that the obfuscation method will be successful in beating the detection classifier. Rather it means that the obfuscation method to some extent impact the method used for building the graphs. Since it is able to impact the graphs it might have an impact on the classifier. However, the impact might also be insignificant. This fully depends on the implementation on the method and which features that are selected and used for the classifier. It is important to note that most of these obfuscation techniques are designed to trick either static or signature-based detection systems. Hence their efficiency, regardless of impact is expected to be lower. In this thesis we will not analyze the efficiency with regard to detection rate. But rather analyze which obfuscation methods that possibly can impact the execution behavior graphs. Hence it is not an application area specific test, but rather an experiment that test the obfuscation resilience of the general method.

5.4.3 Best set of features with regard to detection accuracy and false positives for the different layers

The goal of this research question is to find a set of features that provide the best detection accuracy and false positive rates for all the three layers. This research question

has been taken into account in creation of the dataset. Since creation of three traces will ensure that this research question can be analyzed for all of the different layers.

As mentioned earlier in Section 5.3, the classifier was to be implemented by use of graph matching in SUBDUE. Since SUBDUE is fully automated and able to find patterns on its own, no manual feature selection is needed. Furthermore 10-fold cross validation can be implemented automatically. This will ensure that SUBDUE finds the best features from the dataset and chose subgraphs that exist in malware, but less frequent in software. Furthermore the 10-fold cross validation will ensure that both malware and software has equal impact on the chosen features and thus classifier.

SUBDUE will then use 10-fold cross validation for training and the remaining dataset for testing. When this process is complete it output a confusion matrix from which detection accuracy and false positives and negatives can be analyzed.

Metrics

Since SUBDUE output a confusion matrix the metrics of this experiment will be true positive, true negative, false positive and false negative.

- True positives is the number of correctly classified malware
- True negatives is the number of correctly classified software
- False positives is the number of software incorrectly classified as malware
- False negatives is the number of malware incorrectly classified as software

From these metrics the classification accuracy along with false positive- and false negative rate can be calculated. These are then used as a baseline for discussing the method's efficiency, and comparing detection performed at the different layers.

Results

The no free lunch theorem dictates that whenever there is a classifier that solves a problem, there exists a problem where the classifier fails. Our goal in this experiment is to find classifiers that provide good classification accuracy for malware detection. If one for instance focus on throughput and obfuscation resilience it is likely that there exist other classifiers that would perform better. To summarize, the results provided in this experiment are only focused on providing the best possible detection rate.

6 Experimental setup and results

This chapter includes all the experimental results for each research question. First the reliability testing of information-based dependencies is performed. Then the obfuscation resilience of information-based dependency matching is analyzed. Finally the detection rates of information-based dependency graph matching by use of SUBDUE is evaluated. After the experiments within each research questions summary is provided to list the findings.

6.1 Reliability testing of information-based dependency matching

An introduction to information-based dependency matching was provided in Section 3.3. As mentioned, several reliability issues were discussed. Both from the original paper [11] and [12]. More specifically, both papers discussed issues related to def-use dependencies with regard to non-handle values, such as integers or short strings [11, 12]. Furthermore ordering dependencies were identified as only reliable in conjunction with other methods [12]. False dependencies in Value-based dependencies was also discussed [12].

The purpose of this Section is to shed light on the reliability issues of information-based dependency matching. Furthermore it will complement the previous research with comprehensive analyses of new scenarios in order to fully map the extent of the reliability issues. The motivation for this is to be able to choose a reliable method, which will then be used to for creating dependencies and building graphs that answer the remaining research questions. This was discussed in [11, 12], however not extensively tested. The conclusion was that handles or memory values could be used to create dependencies. However neither of the papers tested all value types in order to investigate how these can be used to create dependencies. Thus, in order to investigate this issue we propose experiments and analyses of the following values: characters, character arrays, integers, structs, classes and boolean values. Furthermore operator overloading will be tested.

6.1.1 Def-use dependencies

A def-use dependence *express that a value output by one system call is used as input to another system call* [11]. More specifically this means that the return value of one function call is used as a parameter for a subsequent function call. There are several challenges with this approach. One is to determine which type of variables and return values that are appropriate for creating dependencies.

Another issue is to determine how far the gap between function call can be in order to infer a dependency. For instance if there are 100 function calls. What are the differences between inferring a dependency between inferred between call 1 and call 100 or is it only subsequent calls that can be dependent.

Boolean example

In Figure 17 we show an example of a function call trace for a very simple boolean function. The figure consist of the code, trace and resulting graph. The trace consist of a set of function call names, which all has parameters and a return value. The graph representation display the function call names represented as vertices with appended timestamp.

```c
#include <stdio.h>
#include <stdbool.h>

bool test(char temp) {
    if(temp=='c')
        return true;
    else
        return false;
}

int main() {
    char c;
    c=getchar();
    if(test(c)==true)
        printf("Character was indeed a %c",c);

    return 0;
}
```

(a) Code

```
02:12:44.849635 __libc_start_main(0x8048416, 1, 0xbfb849c4,
0x8048480, 0x8048470 <unfinished ...>
02:12:44.849695 getchar(0xb8019849, 0x8049ff4, 0xbfb84918,
0x80482e8, 0xb8055ff4 <unfinished ...>
02:12:44.849738 SYS_fstat64(0, 0xbfb847ec, 0xb8055ff4,
0xb8056420, 0xb8056420) = 0
02:12:44.849769 SYS_mmap2(0, 4096, 3, 34, -1)    = 0xb8066000
02:12:44.849798 SYS_read(0, "c\n", 1024)         = 2
02:12:45.467647 <... getchar resumed> )          = 99
02:12:45.467697 printf("Character was indeed a %c", 'c'
<unfinished ...>
02:12:45.467752 SYS_fstat64(1, 0xbfb841c0, 0xb8055ff4,
0xb80564c0, 0xb80564c0) = 0
02:12:45.467783 SYS_mmap2(0, 4096, 3, 34, -1)    = 0xb8065000
02:12:45.467841 <... printf resumed> )           = 24
02:12:45.467864 SYS_write(1, "Character was indeed a c", 24) =
24
02:12:45.467920 SYS_exit_group(0 <no return ...>
02:12:45.467970 +++ exited (status 0) +++
```

(b) Trace

(c) Graph

Figure 17: Boolean function call example

The edges represent a dependency and is labeled by the dependency value. The purpose of the experiment was to see whether functions within the C file could infer dependencies. However as one can see from the output there is only one dependency (after program initialization). This is a dependency inferred by the integer value 24, which is the character c in the program. The input value to character c was also c, however this is not the dependency which is inferred, as the ASCII value of lowercase c is 99, while the hex value is 61. Subsequently this value must be used to reference the character for both functions. What is interesting about this example is the link between the library calls and system calls. Since printf is the output function for library call, while sys_write is the output function used by printf. This serves as a positive example of an actual def-use dependency.

Another interesting fact is that there is no signs of the function test which was defined within the c program. There is no trace of this function in neither library or system calls. This is weird, as we expected to see some sort of check, whether the input character c was similar to the value c. This most likely mean that this checking is performed not by use of library call and system call but implemented directly by use of registers. Although not an argument for or against def-use dependencies it is an important downside of the method in general, as it proves that not all behavior is represented in the trace output, even for software fully implemented by use of library calls.

Struct

The purpose of this example was to test how def-use dependencies relate to objects, more specifically structs.

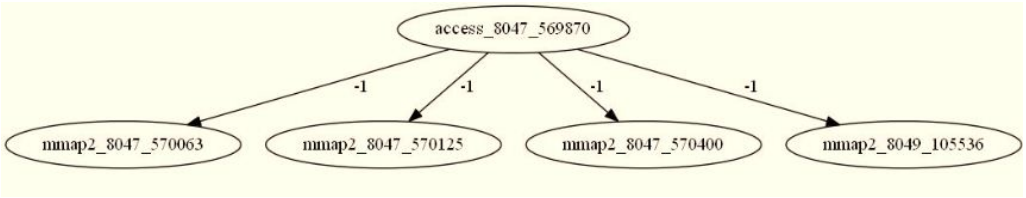
In Figure 18 we can see that there are inferred 4 dependencies based on the value -1. All dependencies exist between the function access and the function mmap. Access is a kernel function which checks a user's permissions to a specific file. Upon success it returns zero, while on error it return -1 [134]. Mmap on the other hand is a kernel function for

```c
#include <stdio.h>

struct test {
    char c;
};

int main() {
    struct test t;
    t.c = getchar();
    printf("%c",t.c);

    return 0;
}
```

(a) Code

```
02:14:07.569870 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT
(No such file or directory)
02:14:07.569903 open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY)
= 3
02:14:07.569934 read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3
\0\1\0\0\0\340g\1"..., 512) = 512
02:14:07.569969 fstat64(3, {st_mode=S_IFREG|0755, st_size=
1425800, ...}) = 0
02:14:07.570005 mmap2(NULL, 1431152, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7ee1000
02:14:07.570032 mmap2(0xb8039000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x158) = 0xb8039000
02:14:07.570063 mmap2(0xb803c000, 9840, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb803c000
02:14:07.570093 close(3)                = 0
02:14:07.570125 mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7ee0000
02:14:07.570153 set_thread_area({entry_number:-1 -> 6,
base_addr:0xb7ee06b0, limit:1048575, seg_32bit:1, contents:0,
read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
02:14:07.570237 mprotect(0xb8039000, 8192, PROT_READ) = 0
02:14:07.570271 mprotect(0x8049000, 4096, PROT_READ) = 0
02:14:07.570298 mprotect(0xb8069000, 4096, PROT_READ) = 0
02:14:07.570323 munmap(0xb803f000, 50913) = 0
02:14:07.570365 fstat64(0, {st_mode=S_IFCHR|0620,
st_rdev=makedev(136, 0), ...}) = 0
02:14:07.570401 mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb804b000
02:14:07.570429 read(0, "c\n", 1024)    = 2
02:14:09.105489 fstat64(1, {st_mode=S_IFCHR|0620,
st_rdev=makedev(136, 0), ...}) = 0
02:14:09.105536 mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb804a000
02:14:09.105572 write(1, "c", 1)       = 1
02:14:09.105601 exit_group(0)           = ?
```

(b) Trace

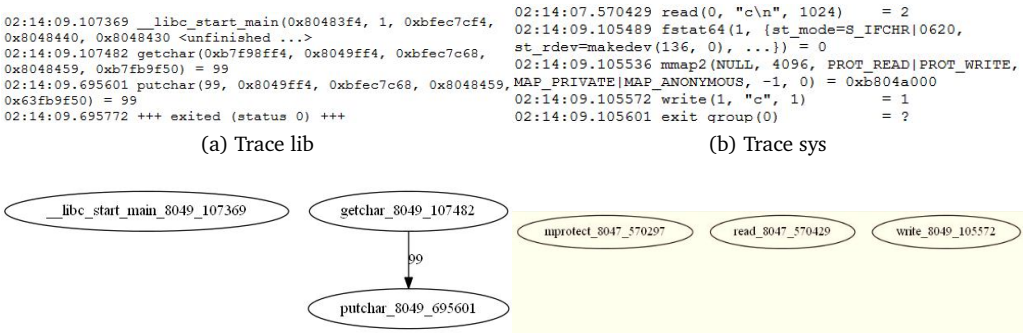


(c) Graph

Figure 18: Struct system call example

mapping files or devices into memory. As one can see from Figure 18b the dependency is inferred by the value -1 which is parameter 5. According to [135], parameter 5 is the file descriptor. And the value -1 simply means that anonymous mapping is used and the mapping is not backed up by any file [135]. Hence a dependency is falsely inferred based on the fact that both functions commonly used -1 as a parameter.

Another interesting fact is shown in Figure 19, where we see output for library and system calls for the same program as above. What is interesting in this scenario is the fact that library calls and their equivalent system calls work in different ways. The reason for this statement is because a dependency is inferred for library calls, but not for system calls. As one can see, getchar and putchar are dependent, but read and write are not. This is interesting as it proves as an example that functions are implemented in different ways at different layers. Which in turn affect the rate at which def-use dependencies can be successfully inferred for that function. Furthermore both these behaviors are captured using function calls, where both library and system call traces are included. Thus this



(a) Trace lib (b) Trace sys

(c) Graph lib (d) Graph sys

Figure 19: Struct system call and library call example

dependency could be detected by both function call and library call trace output.

FileIO

In Figure 20 we have included an example that shows both correct and incorrect use of def-use dependencies. In this example program, the line *Today is the first day of the rest of your life* is read from file. This file is read character by character and output to console the same way. Source is included in Figure 20a. The purpose of this experiment was to test whether one should restrict the sequence for which one can create a dependency. For instance if a dependency must be following the dependent function, or in the last 100 subsequent functions, or simply subsequent, regardless of how far.

If one look at the trace output in Figure 20b, one can see that a handle is returned when fopen is called. This handle ('0x9cc9008') is then used by each fgetc function, as it reads the memory mapped file character by character. This is further visualized in Figure 20d. This example showed how effective memory values are at creating dependencies. Memory values are quite unique, which makes them good for creating dependencies. Furthermore this example shows that dependencies can exist further down the chain of function calls, and not necessarily just the 10 subsequent calls.

In the previous experiment displayed in Figure 19, we saw that integers could be useful for creating dependencies. This experiment on the other hand provides a counter-example, where one can see the impact of false dependencies by use of integer def-use dependencies. Figure 20c emphasize the putchar functions where the first argument is similar to the return value. That is, it gets the character it should write as the first argument, and it returns the same value when completed. This in turn creates a set of false dependencies when several similar characters are printed. This means that every similar character in the line: *Today is the first day of the rest of your life*, is dependent, which is certainly not correct. Graph output which depicts this behavior is provided in Figure 20e. This example proves as an example of how not even high integers can be trusted to be used as def-use dependencies.

Discussion

During these four experiments for def-use dependencies we have found several interesting properties about information-based dependency matching. First we have verified that integers are unsuited, as even high integers easily infer false dependencies, as all numbers a part of the ASCII tables are often utilized. Furthermore we have found that all

```c
#include <stdio.h>
int main() {
    char x;
    FILE *fp;
    fp=fopen("/home/rem/Desktop/infile.txt","r");
    while((x=fgetc(fp))!=EOF) {
        printf("%c",x); }
    fclose(fp);
    fclose(out);
    return 0;
}
```

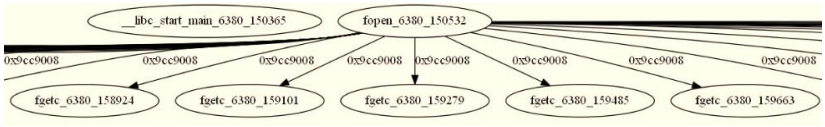
(a) Source FileIO

```
01:46:20.150365 __libc_start_main(0x8048474, 1, 0xbfb831c4,
0x8048500, 0x80484f0 <unfinished ...>
01:46:20.150533 fopen("/home/rem/Desktop/infile.txt", "r") =
0x9cc9008
01:46:20.150807 fgetc(0x9cc9008)                = 'T'
01:46:20.150992 putchar(84, 0x80485b0, 0xbfb83118, 0x804834c,
0xb7f55ff4) = 84
01:46:20.151153 fgetc(0x9cc9008)                = 'o'
01:46:20.151240 putchar(111, 0x80485b0, 0xbfb83118, 0x804834c,
0xb7f55ff4) = 111
01:46:20.151359 fgetc(0x9cc9008)                = 'd'
01:46:20.151446 putchar(100, 0x80485b0, 0xbfb83118, 0x804834c,
0xb7f55ff4) = 100
01:46:20.151537 fgetc(0x9cc9008)                = 'a'
01:46:20.151622 putchar(97, 0x80485b0, 0xbfb83118, 0x804834c,
0xb7f55ff4) = 97
```

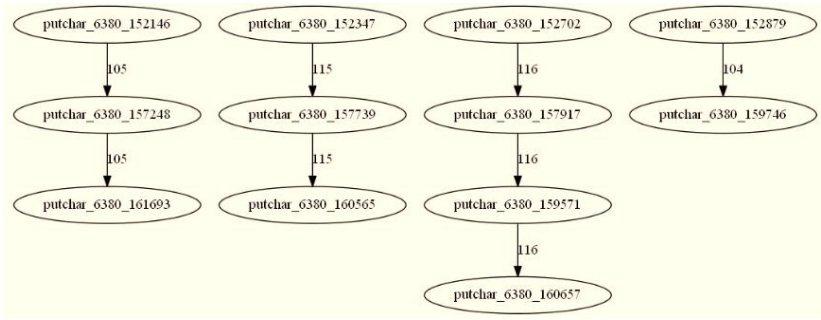
(b) Trace lib1 (correct)

```
01:46:20.159393 putchar(32, 0x80485b0, 0xbfb83118, 0x804834c,
0xb7f55ff4) = 32
01:46:20.159486 fgetc(0x9cc9008)                = 't'
01:46:20.159571 putchar(116, 0x80485b0, 0xbfb83118, 0x804834c,
0xb7f55ff4) = 116
01:46:20.159663 fgetc(0x9cc9008)                = 'h'
01:46:20.159747 putchar(104, 0x80485b0, 0xbfb83118, 0x804834c,
0xb7f55ff4) = 104
01:46:20.159839 fgetc(0x9cc9008)                = 'e'
01:46:20.159924 putchar(101, 0x80485b0, 0xbfb83118, 0x804834c,
0xb7f55ff4) = 101
01:46:20.160015 fgetc(0x9cc9008)                = ' '
01:46:20.160120 putchar(32, 0x80485b0, 0xbfb83118, 0x804834c,
0xb7f55ff4) = 32
01:46:20.160215 fgetc(0x9cc9008)                = 'r'
01:46:20.160347 putchar(114, 0x80485b0, 0xbfb83118, 0x804834c,
0xb7f55ff4) = 114
01:46:20.160429 fgetc(0x9cc9008)                = 'e'
01:46:20.160513 putchar(101, 0x80485b0, 0xbfb83118, 0x804834c,
0xb7f55ff4) = 101
```

(c) Trace lib2 (incorrect)



(d) Graph lib1 (correct)



(e) Graph lib2 (failed)

Figure 20: FileIO library call correct and failed example

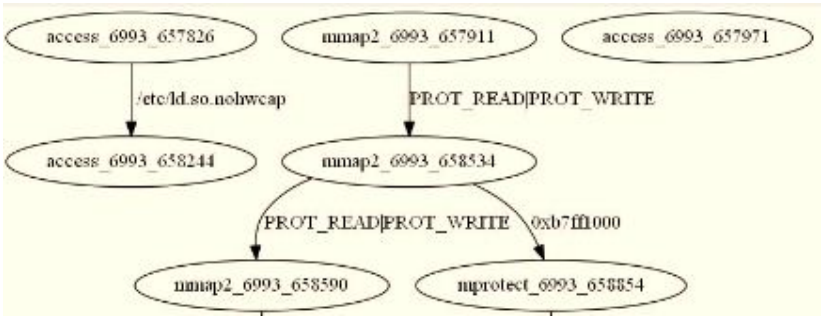
functions have an integer or handle return value. And even though characters are used they are represented through the decimal value of the ASCII table, which makes characters not very useful for def-use dependencies. We have also learned that not all behavior is captured through function calls since if checks and comparisons seem to be implemented directly by use of registers. Furthermore dependencies are inferred based on return values and parameters, thus similar functionality might infer different dependencies for the different layers, depending on the implementation of the call in that layer. We also found that it is hard to put a limit on how subsequent a call should be in order to infer a dependency. We proved in Figure 20d that for instance for fileIO, the limit depends on the size of the file/string, thus making hard limits impossible.

```
#include <stdio.h>
#include <string.h>

int main() {
    char *string;
    char dummy[100];

    gets(dummy);
    strcpy(string,dummy);
    printf("%s",string);

    return 0;
}
```

(a) Code

```
01:56:33.657827 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT
(No such file or directory)
01:56:33.657911 mmap2(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb8004000
01:56:33.657972 access("/etc/ld.so.preload", R_OK) = -1 ENOENT
(No such file or directory)
01:56:33.658031 open("/etc/ld.so.cache", O_RDONLY) = 3
01:56:33.658084 fstat64(3, {st_mode=S_IFREG|0644, st_size=
50913, ...}) = 0
01:56:33.658156 mmap2(NULL, 50913, PROT_READ, MAP_PRIVATE, 3,
0) = 0xb7ff7000
01:56:33.658200 close(3)                = 0
01:56:33.658244 access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT
(No such file or directory)
01:56:33.658304 open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY)
= 3
01:56:33.658359 read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3
\0\1\0\0\0\340g\1"..., 512) = 512
01:56:33.658421 fstat64(3, {st_mode=S_IFREG|0755, st_size=
1425800, ...}) = 0
01:56:33.658487 mmap2(NULL, 1431152, PROT_READ|PROT_EXEC,
MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xb7e99000
01:56:33.658534 mmap2(0xb7ff1000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x158) = 0xb7ff1000
```

(b) Trace



(c) Graph

Figure 21: String system call example

6.1.2 Value dependencies

Value Dependence is a *logic formula expressing the conditions placed on the argument values of one or more system calls* [11]. Hence it is a dependency inferred from similar parameter values from subsequent function calls. This method is useful for finding dependencies from parameter referencing. For instance functions which use character arrays as parameters, which are always reference transferred. For def-use dependencies we showed how unreliable integers can be for inferring dependencies. This is likely to hold true for values as well, given that there are more parameters than return values. Furthermore characters were represented by decimal values, such that we cannot infer dependencies based on single characters. Strings on the other hand are likely to be different. Handle values have proven to be quite unique and good for representing dependencies. However, we should analyze which strings are suitable and their appropriate length. For instance the value *'NULL'* is a commonly used string to represent an empty pointer. Thus this is likely to infer false dependencies. As such the goal of this subsection is to find the appropriate use of strings as value dependencies. To explore this issue we have an experiment which is discussed below.

Experiment 1

To implement the knowledge learned from the experiments in the def-use subsection above, a limit to dependencies based on strings of 8 or more characters was implemen-

ted. This was due to the fact that characters are represented as decimal values, and decimal/integer values are not reliable. Furthermore short strings such as NULL are unreliable, thus a higher threshold was implemented. The source of the program used in this experiment is displayed in Figure 21a, while a subsection of the created dependencies are displayed in Figure 21c. In Figure 21b one can see the trace output belonging to the graph. The dependent traces and values are color encoded. The access functions create a dependency based on the filepath *'/etc/ld.so.nohwcap'*. However the purpose of this function is only to determine access to a file, thus there is likely no real dependency between the functions. Similarly the mmap2 functions infer dependencies based on the parameter PROT_READ PROT_WRITE, which is simply a parameter for mmap2 which maps a file into memory. Thus another example of a false dependency for long string values. Both these examples show that value dependencies easily create false dependencies, since the occurrences of similar parameters are common. Thus we should limit its use by reducing the inferred dependencies to memory values only, since these are quite unique.

6.1.3 Ordering dependencies

Ordering dependence: states that the first function call must precede the second function call [11]. In practice this means that all function calls will be dependent on the previous function call. The goal of this section is to shed light on the issues that may arise from implementing and not implementing this dependency rule.

The complications of implementing this rule was discussed in [12], where it was concluded that subsequent function calls not necessarily are dependent. An example was given where the user input a character, the character is printed to screen, then some text is printed. Obviously the text is not dependent on the previous functions. This is true, however, should not the sequence of execution be implemented as a part of the execution graphs? By not implementing this rule one risk having a graph tree that does not purely represent the execution itself. For instance if the last function call is dependent on the first function call. Then these would be linked in such a way that all the remaining calls are displayed and linked sequentially after the last function call. An example of this can be seen in Figure 21c, where all the dependent function calls are tied to function calls with an early timestamp. Thus the dependent function calls, are moved forward in the execution graph. Subsequently distorting the representation of execution.

A counter argument to this statement could be that sequentially correct models can easily be evaded by implementing dummy functions. Thus obfuscating attack pattern and evading detection. One way to counter this could be by weighting the dependencies, such that ordering dependencies doesn't impact the

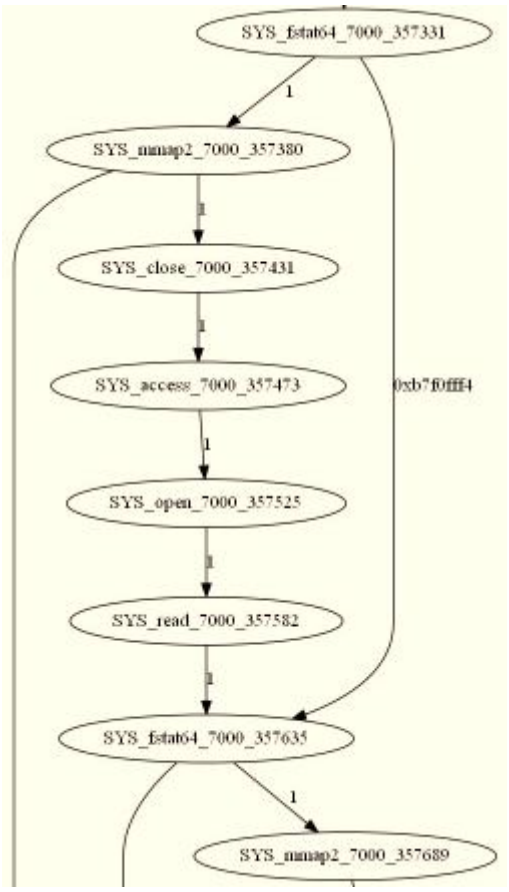


Figure 22: Ordering Dependency example

classifier as much as value and def-use dependencies.
This might be a good idea, seeing that def-use based
dependencies are the hardest to come by, then value-based, then finally ordering, which always occur.

6.1.4 Summary

In this section we have experimented with information-based dependency matching and the three rule-types it use. The goal was to find how reliable each rule-type was, and for which variables they could utilize to create dependencies. Furthermore we have found other interesting findings and reliability issues that might impact the method and implementation in thesis. To summarize the results of the experiments performed, we will in this subsection list the findings. First the finding is listed, then the impact on implementation or method is listed below.

- Characters are represented by decimal values and must be considered integers (Ref Section 6.1.1).
  Impact: No single characters are used to find dependencies.
- Use of integers are unreliable for dependency matching (Ref Section 6.1.1).
  Impact: No integers are used to find dependencies.
- If checks and value comparison might be implemented by use of registers as no trace of such activity can be captured using function calls (Ref Section 6.1.1).
  Impact: No impact on implementation, but a major downside, as it proves that not all behavior can be captured using function calls.
- Functions import and export data using different mechanisms. Thus certain dependencies can only be detected if the functions utilize such behavior. This leads to different dependencies based on different functions for both user-mode and kernel-mode (Ref Section 6.1.1).
  Impact: Kernel-mode and user-mode dependencies and thus graph behavior will be different, even for the same application. Which is interesting as some malware might only be detected at specific layers.
- Use of strings are unreliable for dependency matching, with the exception of handles and memory values (Ref Section 6.1.2).
  Impact: Only memory values, I.E. strings that start with 0x are used to find dependencies.
- Ordering dependencies infer false dependencies, however prove necessary to provide accurate graph behavior (Ref Section 6.1.3).
  Impact: Ordering dependencies are implemented.
- Weighting dependencies based on rule-type might be a good idea to correctly represent the uniqueness of a dependency (Ref Section 6.1.3).
  Impact: Not implemented, but could be interesting for further work.
  Argument: Might be a great idea, given that memory values are environment specific and shouldn't be used for detection. Thus a weighted approach with 1,2,3 might be more suitable.

```c
#include <stdio.h>

int main() {
    printf("Hello world!!!");
    return 0;
}
```

(a) Sample Code

```
11:05:11.951366 __libc_start_main(0x80483c4, 1, 0xbf96afa4,
0x8048400, 0x80483f0 <unfinished ...>
11:05:11.952077 printf("Hello world!!!")         = 14
11:05:11.953227 +++ exited (status 0) +++
```

(b) Sample trace

```c
#include <stdio.h>

int main() {
    int i=0;
    printf("Hello world!!!");
    printf("%i",i);
    return 0;
}
```

(c) Dead Code Insertion

```
11:05:55.403927 __libc_start_main(0x80483c4, 1, 0xbfbc31e4,
0x8048420, 0x8048410 <unfinished ...>
11:05:55.404657 printf("Hello world!!!")         = 14
11:05:55.405368 printf("%i", 0)                  = 1
11:05:55.406078 +++ exited (status 0) +++
```

(d) Dead Code Insertion Trace

Figure 23: Dead code insertion example

Variable type	Unreliable	Comment
Integers	Yes	-
Characters	Yes	Characters are represented by ASCII numbers and thus integers by definition
Boolean values	Yes	represented as integers
Strings	Yes	With the exception of memory values

Table 3: Variable type reliability

6.2 Obfuscation resilience of information-based dependency matching

The goal of this section is to measure the obfuscation resilience of information-based dependency matching. The information-based dependency matching implementation is based on the previous section. To measure the obfuscation resilience we perform a set of experiments against the obfuscation methods discussed in Section 2.6.

The experiments are like the previous section based on synthesize testing. First we create a simple C application and trace output. Then we implement the obfuscation method, and trace output. The trace logs are then compared in order to find whether the obfuscation technique had an impact on the method.

6.2.1 Dead code insertion

For this experiment we set up a simple hello world program and then implemented an obfuscated version where we added another printf function that print a character to screen. The code and trace output can be viewed in Figure 23. As one can see from the trace ouput another printf line is added to the output. This shows that dead code insertion can be detected using function calls. It is important however, to note that not all dead code insertion can be detected. For instance arithmetic operations and if checks as discussed earlier will not be detected on function call traces as they are likely to be implemented directly by use of registers.

```
section .text
    global _start

_start:
    mov edx,len
    mov ecx,msg
    mov ebx,1
    mov eax,4
    int 0x80

section .data
msg db  'Hello world!',0xa
len equ $-msg
```

(a) Sample Hello World

```
section .text
    global _start

_start:
    mov eax,len
    mov ecx,msg
    mov ebx,1
    mov edx,eax
    mov eax,4
    int 0x80

section .data
msg db  'Hello world!',0xa
len equ $-msg
```

(b) Register Reassignment

```
13:14:06.609073 execve("./sample_hello", ["./sample_hello"],
[/* 35 vars */]) = 0
13:14:06.611555 write(1, "Hello world!\n", 13) = 13
13:14:06.611925 --- SIGSEGV (Segmentation fault) @ 0 (0) ---
13:14:06.612548 +++ killed by SIGSEGV +++
```

(c) Sample trace

```
14:09:18.306457 execve("./obfuscated_register_reassignment",
["./obfuscated_register_reassignme"...], [/* 35 vars */]) = 0
14:09:18.308114 write(1, "Hello world!\n", 13) = 13
14:09:18.308436 --- SIGSEGV (Segmentation fault) @ 0 (0) ---
14:09:18.308805 +++ killed by SIGSEGV +++
```

(d) Register Reassignment Trace

Figure 24: Register Reassignment example

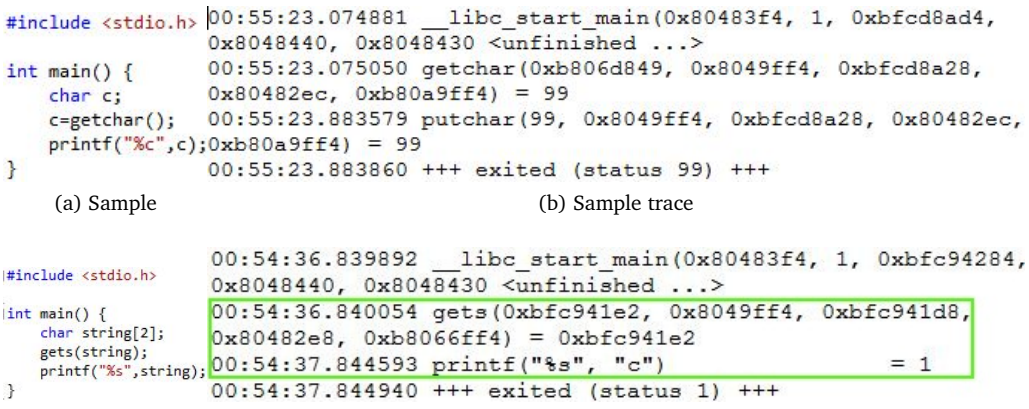
6.2.2 Register reassignment

In this experiment we test the impact of the register reassignment obfuscation technique, which is based on swapping of registers used by live variables. To do this we created a very simple hello world program in assembly. The regular program can be seen in Figure 24a, while the obfuscated example is provided in 24b. By comparison, one can see that the only difference is a reassignment of registers, which is outlined in green. Trace output for both programs is provided below in Figure 24c and 24d. From the trace output one can see that there are no differences. As a result, the register reassignment method have no impact on our method.

Another interesting fact about assembly is that it is very low level programming, which bypasses regular library calls. Thus tracing the execution of assembly-based programs will always result in an empty trace output. For system calls on the other hand we will get a successful trace, as can be seen in Figure 24c and 24d. This is important, as assembly-based programs only can be traced and thus detected by system call-based detection.

6.2.3 Code Substitution

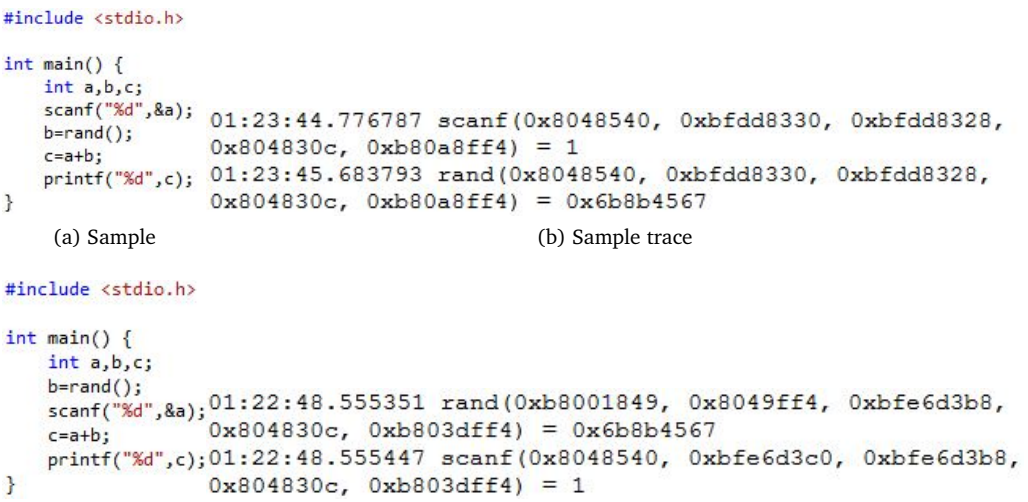
In Figure 25 we have included an example of code substitution. The figure consists of code and trace for both sample and obfuscated code. The experiment is pretty simple, as it only replaces the getchar function with a gets function, which is equivalent in functionality for our purpose. The resulting trace output in Figure 25d is as expected different. However, there is another interesting finding, which is that the succeeding output function changed as well. Putchar changed to printf. This might be simply because that single



(a) Sample (b) Sample trace

(c) Code Substitution (d) Code Substitution Trace

Figure 25: Code Substitution example



(a) Sample (b) Sample trace

(c) Code Reordering (d) Code Reordering Trace

Figure 26: Code Reordering example

characters are printed using putchar, while strings are printed using printf. However, it shows that by changing one function, succeeding functions might change as a result.

6.2.4 Code reordering

The code reordering obfuscation technique deal with the reordering of functions in order to change the sequence of execution. To create this scenario we created a simple program where an integer is given by user, then b is randomly generated, while c consist of the sum. Both the acquisition of a and b are independent and could be in any order, as long as they occur before c. Thus the obfuscated version is simply a change of sequence. The experimental code and trace is provided in Figure 26. From the result one can see that the sequence of functions is changed.

6.2.5 White space and comment randomization

We mentioned in Section 2.6.5 about white space randomization and comment randomization that these were techniques that affect code that is parsed pre-assembly or pre-

```c
#include <stdio.h>

int main() {
    char string[]="Hello world!";
    printf("%s",string);
    return 0;
}
```

(a) Sample

```
02:29:41.891598 __libc_start_main(0x8048414, 1, 0xbfeed4e4,
0x8048490, 0x8048480 <unfinished ...>
02:29:41.891749 printf("%s", "Hello world!")      = 12
02:29:41.895156 +++ exited (status 0) +++
```

(b) Sample trace

```c
#include <stdio.h>

int main() {
    char string[]="Hello world!";
    //THIS IS A COMMENT then whitespace

    printf("%s",string);
    return 0;
}
```

(c) Obfuscated code

```
02:29:21.531738 __libc_start_main(0x8048414, 1, 0xbf856644,
0x8048490, 0x8048480 <unfinished ...>
02:29:21.531896 printf("%s", "Hello world!")      = 12
02:29:21.535227 +++ exited (status 0) +++
```

(d) Obfuscated Trace

Figure 27: Whitespace and comment randomization example

execution. For instance for JavaScript, where detection typically is performed before the browser parse the JavaScript. Reason for this statement is that the parsers ignore such comments and white space. However, in order to verify and test this, we implemented a simple experiment. This code and trace for both sample and obfuscated example is given in Figure 27. The resulting trace is as expected similar.

6.2.6 String obfuscation

The idea behind string obfuscation is that there exist several ways of representing the string. A string might be encoded differently or separated by a range of variables, then concatenated before execution. Thus there might be several examples of this technique that both impact and doesn't impact our method. However, since obfuscation techniques that impact our method are important we sought to find proof of this in our experiment. The experiment can be viewed in Figure 28, where both code and trace is provided for both sample and obfuscated example. The trace output for the obfuscated example is only a selected section of the trace. However it shows as an example of how execution is different, as it executes every character by itself and not the complete string. Our example is however, dependent on how the string is used. But that is likely to be true for malware as well. Because of this we conclude that string obfuscation impact our method.

6.2.7 Variable and function name randomization

Variable and function randomization is a technique used to fool detection systems and human analysts, by reassigning variables and functions. To test this we created an experiment where a string hello world is transferred through variables and a function in order to be printed. The output of both programs should be "“Hello World!"". The experimental code and trace for sample and obfuscation technique can be found in Figure 29. From the figure one can see that they have the same trace output, and that reassigning the variables and functions had no effect on the execution. This is logical, as no additional commands are executed.

6.2.8 Opaque predicates

Opaque predicates is an obfuscation technique where one uses statements whose result is known in advance, but not obvious. These might both impact and not impact the trace output depending on method used. For instance, one might include a complex

```c
#include <stdio.h>

int main() {
    printf("HELLO WORLD!");
    return 0;
}
```

(a) Sample

```
04:36:29.558173 __libc_start_main(0x80483c4, 1, 0xbf92d734,
0x8048400, 0x80483f0 <unfinished ...>
04:36:29.558334 printf("HELLO WORLD!")           = 12
04:36:29.558548 +++ exited (status 0) +++
```

(b) Sample trace

```c
#include <stdio.h>

int main() {
    int i;
    char c[]="H@eXl3loo/W#o%rllpd";
    for(i=0; i<=19; i+=2) {
        printf("%c",c[i]);
    }
    return 0;
}
```

(c) Obfuscated code

```
04:36:17.822348 __libc_start_main(0x8048414, 1, 0xbff04d04,
0x80484b0, 0x80484a0 <unfinished ...>
04:36:17.822508 putchar(72, 0x8049ff4, 0, 0x58654048,
0x6f6c336c) = 72
04:36:17.822653 putchar(101, 0x8049ff4, 2, 0x58654048,
0x6f6c336c) = 101
04:36:17.822735 putchar(108, 0x8049ff4, 4, 0x58654048,
0x6f6c336c) = 108
04:36:17.822861 putchar(108, 0x8049ff4, 6, 0x58654048,
0x6f6c336c) = 108
```

(d) Obfuscated Trace

Figure 28: String obfuscation example

```c
#include <stdio.h>
int main() {
    printf("Hello world!");
    return 0;
}
```

(a) Sample

```
02:58:15.377529 __libc_start_main(0x80483c4, 1, 0xbfa26814,
0x8048400, 0x80483f0 <unfinished ...>
02:58:15.377678 printf("Hello world!")           = 12
02:58:15.377938 +++ exited (status 0) +++
```

(b) Sample trace

```c
#include <stdio.h>
void xza(char x[13]) {
    printf("%s",x);}
int main() {
    char c[13]="Hello World!";
    xza(c); return 0;}
```

(c) Obfuscated code

```
02:58:03.258015 __libc_start_main(0x804842f, 1, 0xbfa56834,
0x80484a0, 0x8048490 <unfinished ...>
02:58:03.258283 printf("%s", "Hello World!")     = 12
02:58:03.258579 +++ exited (status 0) +++
```

(d) Obfuscated Trace

Figure 29: Variable and function randomization example

calculation. Which is calculated by use of registers, and thus not traced. However, one might also include other functions that will result in an execution trace. To prove this we implemented an experiment where the time function is used to extract the seconds since 1.1.1970. This time value is then used to check if there has been 30000 hours since that date. Which to an analyst is not obvious, but always true. The experiment can be viewed in Figure 30, and as one can see from the trace output, there is a difference in execution.

6.2.9 Inlining and outlining

Inlining was earlier described as a compiler optimization technique where a call to a function is replaced with that functions entire code. This resulted in better performance, as overhead of calling function was removed. An opposite technique is called outlining, where code is placed in functions, rather than duplications. This has the beneficial effect that the code looks more compact. Both techniques, impact the code. However, will it impact the execution trace as well, and subsequently our method? The goal of this experiment is to find the answer to this question. To do this we implemented a simple experiment, where an integer is typed by user then printed to screen. This is performed twice, to illustrate a proper example of both inlining and outlining. That is, functions are typically implemented for code that occurs more than once. The experiment can be

```c
#include <stdio.h>
int main() {
    printf("Hello world!");
    return 0;
}
```

(a) Sample

```
03:37:11.469112 __libc_start_main(0x80483c4, 1, 0xbf993fa4,
0x8048400, 0x80483f0 <unfinished ...>
03:37:11.469261 printf("Hello world!")           = 12
03:37:11.469515 +++ exited (status 0) +++
```

(b) Sample trace

```c
#include <stdio.h>
#include <time.h>
int main() {
    time_t s;s=time(NULL);
    if((s/3600)>30000) {
        printf("Hello world!");}
    return 0;}
```

(c) Obfuscated code

```
03:36:57.868774 __libc_start_main(0x80483f4, 1, 0xbfef5d04,
0x8048450, 0x8048440 <unfinished ...>
03:36:57.868931 time(NULL)                       = 1333708617
03:36:57.869033 printf("Hello world!")           = 12
03:36:57.872298 +++ exited (status 0) +++
```

(d) Obfuscated Trace

Figure 30: Opaque example

```c
#include <stdio.h>
int main() {
    int a;
    scanf("%d",&a);
    printf("%d",a);
    scanf("%d",&a);
    printf("%d",a);
    return 0;}
```

(a) Inlining

```
02:50:58.510036 __libc_start_main(0x80483f4, 1, 0xbfcddaf4,
0x8048470, 0x8048460 <unfinished ...>
02:50:58.510261 scanf(0x8048520, 0xbfcdda50, 0xbfcdda48,
0x80482e8, 0xb7fb0ff4) = 1
02:51:01.257504 printf("%d", 10)                 = 2
02:51:01.257624 scanf(0x8048520, 0xbfcdda50, 0xbfcdda48,
0x80482e8, 0xb7fb0ff4) = 1
02:51:04.350342 printf("%d", 10)                 = 2
02:51:04.350535 +++ exited (status 0) +++
```

(b) Inlining trace

```c
#include <stdio.h>
int outline() {
    int a;
    scanf("%d",&a);
    printf("%d",a);}
int main() {
    outline();
    outline();
    return 0;}
```

(c) Outlining

```
02:51:25.268605 __libc_start_main(0x8048422, 1, 0xbfd75394,
0x8048460, 0x8048450 <unfinished ...>
02:51:25.268759 scanf(0x8048510, 0xbfd752e4, 0xbfd7575a,
0xb7e5adae, 0xb7f09849) = 1
02:51:25.891931 printf("%d", 10)                 = 2
02:51:25.892096 scanf(0x8048510, 0xbfd752e4, 0xbfd7575a,
0xb7e5adae, 0xb7f09849) = 1
02:51:26.580381 printf("%d", 10)                 = 2
02:51:26.580577 +++ exited (status 0) +++
```

(d) Outlining trace

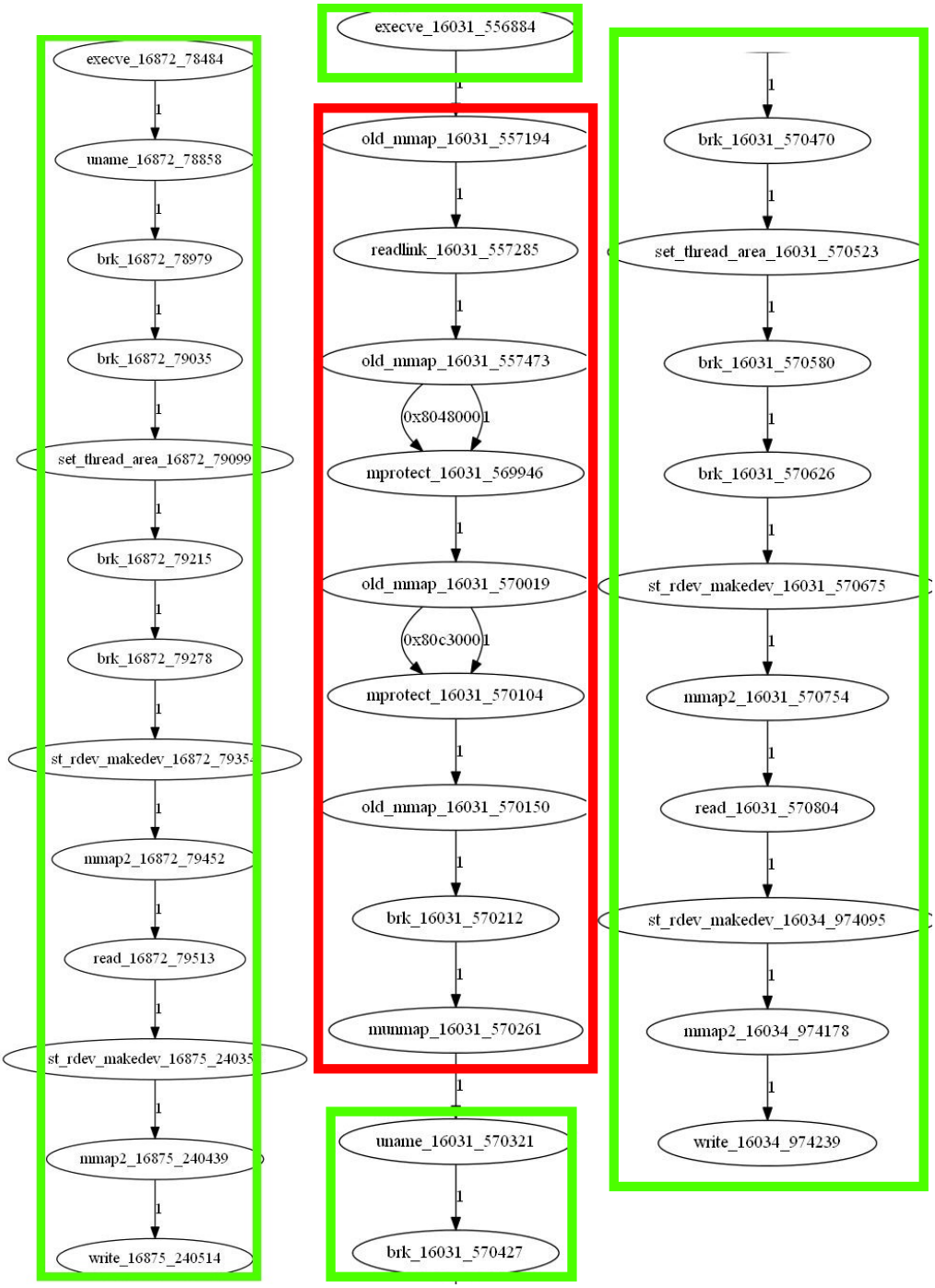
Figure 31: Inlining and outlining example

viewed in Figure 31. From the trace output in Figure 31b and 31d from both inlining and outlining, one can see that the execution behavior is the same. Thus the technique has no impact on our method.

6.2.10 Packers

In Section 2.6.4 of the literature study we discussed packers and how they are implemented to obfuscate malware. In this section we seek to see whether packers are effective against our method, or if we can detect the behavior crated by the packers. To carry out this experiment we decided to implement a packer on one of our previous experiments, namely the string experiment depicted in Figure 21. The string program takes a string as input then output the same string. To implement the packer we decided to use UPX [42], since it is able to pack ELF files, which are the executable files for Linux. There is however a bug in UPX, such that our program had to be compiled statically, which unfortunately could not be traced by ltrace. Subsequently we only have the system call traces for the

packed and unpacked program.



(a) Graph Unpacked (b) Graph Packed

Figure 32: Packer Example

For this experiment we chose to include dependency graphs rather than trace output, to help visualize how easy it is to identify an obfuscation routine using our method. The experiment can be seen in Figure 32, where the unpacked trace is viewed in Figure 32a,

while the packed trace had to be split it in two due to lack of space and is displayed in Figure 32b. The figures are color encoded in such a way that the green is similar for both, while red system calls exist purely for packers. From this behavior we can see that our method is able to trace and detect packers, and not only see the program's functionality, but also identify the obfuscated packing routine. In conclusion, the packed example has an impact, but can also easily be identified using information based dependency matching for function calls.

Obfuscation technique	Impact	Comment
Packers	Yes	-
Dead Code Insertion	Yes	There exist "‘dead code"’ which will not impact as well, such as arithmetic operations
Register reassignment	No	Assembly could only be traced by system calls
Code substitution	Yes	Succeeding functions might change as a result
Code reordering	Yes	Only affect sequence of calls
White space randomization	No	Ignored by parser
Comment randomization	No	Ignored by parser
String obfuscation	Yes	Likely to be highly dependent on use of string
Variable and function name randomization	No	
Opaque Predicates	Yes	There exist "‘opaque predicates"’ which will not impact as well, such as arithmetic operations
Inlining and outlining	No	

Table 4: Obfuscation Matrix

6.2.11 Summary

To summarize the obfuscation resilience experiments, a total of 6 out of 11 methods had an impact on our method. However, it is important to note that most of these techniques are designed to trick signature-based or static detection systems. Hence their effectiveness on information-based dependency matching for function calls has never been tested. Due to the fact that detection mechanisms based on machine learning utilize both the known good and known bad, it is hard to predict the obfuscation techniques effectiveness. The obfuscation techniques might both be effective and counter-effective. Some techniques might obfuscate the sequence of execution in such a way that the malware is classified as software. However, the distinctive patterns of each obfuscation technique might also contribute to making detection easier. This was easily seen in the packer experiment in Section 6.2.10. In conclusion, it is only the behavioral-based obfuscation techniques that has an impact on our method. However, the effectiveness of these should be measured by use of a controlled dataset. This is only possible where one know exactly what malware and software utilize the different techniques. Furthermore, the results show that 5 out of 11 techniques does not have an impact at all, which is good, seeing as this was one of the motivations for doing detection at this layer.

6.3 Classification accuracy for different layers

The goal of this section is to analyze the differences of performing detection at the different layers. Experiments for analyzing detection rate will be carried out using information-based dependency graph matching for library calls, system calls and function calls (hybrid). Each subsection will describe the detection rate and subfigure for each of the methods used for detection.

6.3.1 System calls

The classification accuracy of system calls is 95,8%, 95,8% and 98,9% for MDL, Size and Set Cover, respectively. Seeing as no form of manual feature selection or expert knowledge was provided, these rates seem unusually high. It is however the methods job to parse graphs, learn and find patterns that discriminates well. This seems to have worked well for this particular dataset. For all three methods there are three false positives, while both MDL and Size has eight false negatives. Since it is the Set cover method that discriminates best we will focus on explaining the results of this method.

The confusion matrix for set cover is depicted in Table 5c. As mentioned there are only three false positives. More specifically these are the programs Sed, hostname and dnsdomainname that were misclassified as malware. To fully understand how the classifier is able to achieve this classification accuracy we must analyze the subgraph used for detection. For completeness we have included the graph in Figure 33. As one can see from this figure, this is actually a subgraph of the memory initialization routine. The program execution start at the execve call, which is followed by a brk, access, mmap2, access, open, fstat64, mmap2 and finally a munmap call. The left side of the graph shows mmap2, mprotect, mprotect and finally munmap. From the figure one can see that most of the dependencies are sequence based. However there are also an ordering-based dependency and a def-use dependency, denoted by two and three, respectively.

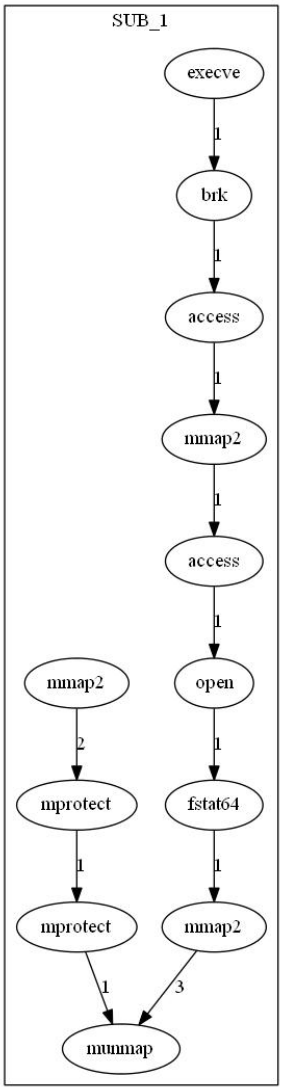


Figure 33: Set Cover Subgraph

After analyzing a set of malware one can see that this substructure exist in all of the malware. This would be very unusual, if it was malware behavior. However, since it actually is the program initialization and loading into memory it is quite feasible. Furthermore, the substructure only occur in 3/75 software. Which begs the question of how this is feasible, and whether there really is such a big difference in program initialization between software and malware.

To answer this question one must understand how programs are loaded into memory, and what aspects of a program that might affect this initialization. After surveying the issue we came up with the following possible reasons: programming language, wrapper or packer functionality, compiler, compiler flags and package dependencies. Furthermore, if one look at the subgraphs of the

MDL		
	Classified as	
Correct class	Positive	Negative
Positive	182	8
Negative	3	72
Classification accuracy		95,8%

(a) MDL

Size		
	Classified as	
Correct class	Positive	Negative
Positive	182	8
Negative	3	72
Classification accuracy		95,8%

(b) Size

Setcover		
	Classified as	
Correct class	Positive	Negative
Positive	190	0
Negative	3	72
Classification accuracy		98,9%

(c) Set Cover

Table 5: System call confusion matrix

software one will see that there is a difference in the mprotect sequence. The most important difference between malware and software is that the malware only have two succeeding mprotect calls, while software has three. This can be seen in Figure 34b and Figure 34a. This point to a difference in memory protection for the different classes. But since both malware and software was executed in a similar fashion in the same Ubuntu environment the difference has to be within the binaries, and not the environment. This realization leads to the discovery of the implementation of Stack Guard in Linux compilers. It turns out that all malware is either compiled with either an old compiler or without stack protection as an alternative. While the three misclassified software are likely to be compiled with an old version of GCC. This in turn leads to a difference in initialization, as the binaries compiled with stackguard will have more memory protection. Since this is kernel mode detection and not user mode detection, such behavior is traced by the system calls.

To analyze whether stack protection was the reason for the difference between the classes we set up an experiment where we compiled a hello world c application with GCC v4.6.3 and GCC v2.9.5. The latest version includes stack protection while the oldest does not. The difference between the resulting graphs can be seen in Figure 35. As one can see from the picture the graph with stack protection contain one extra mprotect system call. The graph that was based on the old compiler would have resulted in a false positive, while the one compiled with the newest compiler would not. Hence it is highly

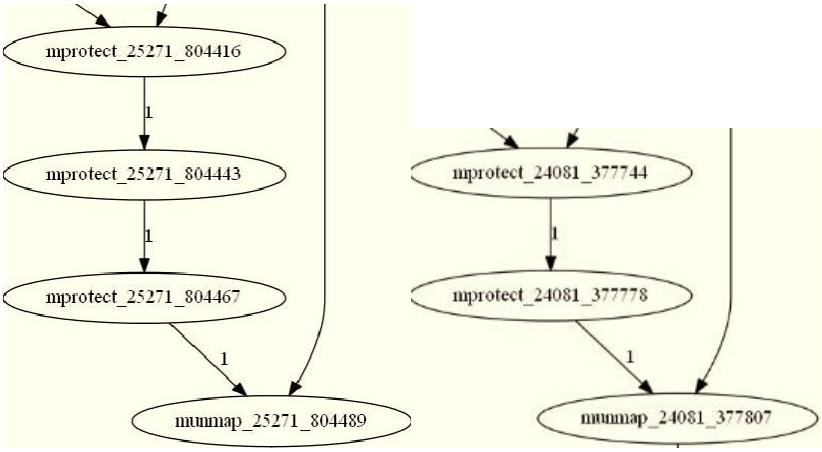
```
09:08:34.053006 mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7ed5000
09:08:34.053061 set_thread_area({entry_number:-1 -> 6,
base_addr:0xb7ed56b0, limit:1048575, seg_32bit:1, contents:0,
read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
09:08:34.053177 mprotect(0xb802e000, 8192, PROT_READ) = 0
09:08:34.053233 mprotect(0x804a000, 4096, PROT_READ) = 0
09:08:34.053286 mprotect(0xb805e000, 4096, PROT_READ) = 0
09:08:34.053335 munmap(0xb8034000, 50913) = 0
```

(a) Bzip2recover(Software)

```
05:47:10.759016 mmap2(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7dfd000
05:47:10.759044 set_thread_area({entry_number:-1 -> 6,
base_addr:0xb7dfd6b0, limit:1048575, seg_32bit:1, contents:0,
read_exec_only:0, limit_in_pages:1, seg_not_present:0,
useable:1}) = 0
05:47:10.759110 mprotect(0xb7f56000, 8192, PROT_READ) = 0
05:47:10.759141 mprotect(0xb7f86000, 4096, PROT_READ) = 0
05:47:10.759166 munmap(0xb7f5c000, 50913) = 0
```

(b) Backdoor.Linux.Blackhole(Malware)

Figure 34: Malware and software difference



(a) Stack protection (b) No stack protection

Figure 35: Stack protection difference

likely that this is the reason for the difference in execution.

Using the memory initialization for classifying malware is not beneficial. Simply because a binary does not utilize stack protection does not mean that the binary is malicious. It could merely be an outdated compiler or removed by other reasons. However, this is not to say that the classification method was successful. The goal of subdue is to find subfigures that discriminate well. Which in this experiment worked very well, since it got a classification accuracy of 98,9%. The only downside was the dataset, since all malware was compiled with either old compilers or without stack protection.

This experiment proves as an example that feature selection and expert knowledge is invaluable when creating a classifier. Since one must always understand the result and how the classifier actually works. By using feature selection one can guide the creation of the classifier by choosing features that are both relevant and non-redundant. In this experiment the features would be the chosen subgraphs.

6.3.2 Library calls

The classification accuracy for library calls is 32,5%, 32,5% and 91,7% for MDL, Size and Set cover, respectively. This is a vast difference, but not surprising, seeing that the two first methods seek to find subgraphs that compress the best, while the Set cover method is best at finding discriminating patterns. From the confusion matrixes in Table 6 one can see that both MDL and Size has 179 false negatives and 0 false positives, while the Set cover method has 22 false negatives and 0 false positives. The Set cover method result is quite extraordinary because it has a very high classification accuracy, but a zero false positives.

Initial analysis of the MDL and Size methods showed that they had both chosen a large set of printf calls for subgraphs. This is reasonable as printf was used to output large sections of text. However, since these printf subgraphs does not occur in all malware they did not prove to be a good classifier. This can be rectified by utilizing feature selection before classification. However, due to the amount of work this would result in, we will not perform feature selection in this thesis. Instead we will focus on the results of the Set cover method to explain the classifier behavior.

MDL		
	Classified as	
Correct class	Positive	Negative
Positive	11	179
Negative	0	75
Classification accuracy		32,5%

(a) MDL

Size		
	Classified as	
Correct class	Positive	Negative
Positive	11	179
Negative	0	75
Classification accuracy		32,5%

(b) Size

Setcover		
	Classified as	
Correct class	Positive	Negative
Positive	168	22
Negative	0	75
Classification accuracy		91,7%

(c) Set Cover

Table 6: Library call confusion matrix

The Set cover method created a subgraph that consisted of __register_frame_info and __deregister_frame_info. Unfortunately, these features seem to be closely related to the compiler issue we discussed in the previous section. According to [136] these calls are used for exception handling. Further literature [137] indicate that these might be compiled into the binary by the compiler to provide exception handling. As such it is possible that these features stem from an old compiler, which does not exist in binaries compiled with newer compilers. This statement is strengthened by the fact that none of the software included this library call.

To analyze whether this could be true or not we implemented an experiment to check whether a new version of GCC would have the same trace output for library calls as an old version. Then a total of five malware was downloaded from Exploit-DB [138]:

- Linux Kernel 2636 exploit
- Linux Kernel 2636 canbcm exploit
- Linux Kernel 2637 exploit
- Linux kernel emulation exploit
- Tivoli IBM exploit

```
02:49:24.290113 __libc_start_main(0x80495fa, 1, 0xbfdce3d4,
0x8049640, 0x8049630 <unfinished ...>
02:49:24.290246 puts("[+] looking for symbols...") = 27
02:49:24.290395 fopen("/proc/kallsyms", "r")     = 0x9393008
```

(a) GCC 4.6.3

```
02:50:59.158276 __libc_start_main(0x80495e0, 1, 0xbfcd12e4,
0x8049610, 0x8049600 <unfinished ...>
02:50:59.158662 __register_frame_info(0x804a000, 0x804a180,
0xbfcd2eff, 0xb7fb8dae, 0xb8067849) = 0x804a000
02:50:59.158838 printf("[+] looking for symbols...\n") = 27
02:50:59.159167 fopen("/proc/kallsyms", "r")     = 0x9fed008
```

(b) GCC 2.9.5

Figure 36: Register frame info

All five malware were compiled with both GCC version 4.6.3 and GCC version 2.9.5, then traced. For all traces of GCC 2.9.5 we found both __register_frame_info and __deregister_frame_info in the library call traces. These were not present in any of the trace

files for GCC 4.6.3. An extract of the trace can be viewed in Figure 36, where one can see that Figure A 36a for GCC 4.6.3 does not include register frame info, but Figure B 36b for GCC 2.9.5 does. Because of this, we argue that the subgraph used for detection for library calls is not malicious behavior, but a result due to compiler difference.

6.3.3 Function calls

The classification accuracy for Function calls are 92,8% for Set Cover. Furthermore it had three false positives and 16 false negatives. The full results can be viewed in Table 7. SUBDUE was not able to finish for MDL and Size, which proves as an example of how high the computational complexity of graph matching really is. While the library calls with set cover used only a few minutes to complete learning of subgraphs, the function call dataset with MDL and Size used over two days without completing the learning process. This is not only because MDL and Size are slower than set cover, but because function calls have more traces than library calls. This is important as the runtime and complexity grows rapidly with an increase in number of function calls. An illustration of this can be seen in Appendix D, where the runtime of the trace parser is tested.

Function calls are a hybrid layer consisting of both library calls and system calls. As such it is expected that one will see similarities in detection for either one of the layers or both. This assumption turns out to be true, as the subgraph used for detection for function calls are similar to that of system calls. The subgraph can be viewed in Figure 37. The subgraph consists of 13 vertices and 13 edges, that all are based on sequence-based dependencies. The graph consists of the memory initialization routine, up until the start of the program (__libc_start_main). The similarity to system calls is reflected in the high classification accuracy and the low amount of false positives as well. However, due to the fact that the subgraph was based on the memory initialization routine, and this routine was proven unreliable in Section 6.3.1, the graph and hence results are not reliable.

Setcover		
	Classified as	
Correct class	Positive	Negative
Positive	174	16
Negative	3	72
Classification accuracy		92,8%

Table 7: Function call Set cover confusion matrix

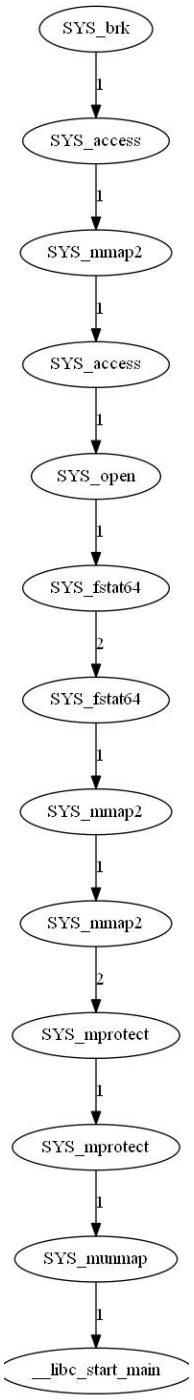


Figure 37: Set Cover Subgraph

6.3.4 Summary

To summarize the classification accuracy for the different layers, the best results were 98,9%, 91,7% and 92,8% by use of set cover for system calls, library calls and function calls, respectively. The full results can be viewed in Table 8, where the classification accuracy of all the methods for the different layers is provided. The fact that set cover was best is not unexpected seeing as it was the best method for discriminating between classes. Unfortunately the results were due to subgraphs that didn't measure anomalous/malicious behavior, but rather compiler differences. This was the case for both user mode execution and kernel mode execution. Kernel mode was affected since no stack protection was used, while user mode by old GCC compiler exception handling routine. This is not to say that the method was not successful in finding discriminative patterns, however it is challenging to say whether this was because of the dataset or by design. The results give strong indications that careful selection of subgraphs is needed. Such that a set of subgraphs are found and only the ones responsible for anomalous/malicious behavior are chosen. In order to do this, knowledge of the environmental specific variables is needed.

	MDL	Size	Set Cover
System calls	95,8%	95,8%	98,9%
Library calls	32,5%	32,5%	91,7%
Function calls	-	-	92,8%

Table 8: Summary of classification accuracy

7 Implications and discussions

In Chapter 6 the experimental results of this thesis was presented. The experiments focused on the reliability issues of information-based dependencies, obfuscation resilience and classification accuracy of the different layers. This chapter provides discussions of the implications as well as summary of the thesis. First the theoretical and practical implications of the obtained results are discussed. Then the thesis, its research questions and findings is summarized at the end of this chapter.

7.1 Theoretical considerations

This thesis sought to improve upon the theoretical foundations of information-based dependency matching. This was performed by analyzing both the reliability issues and obfuscation resilience of the method. Furthermore the potency of the technique has been analyzed with regard to its ability to classify between malware and software. The purpose of this section is to reflect upon the theoretical implications of the findings.

From the experimental studies regarding reliability performed in Section 6.1, it was found that only strings that represented memory values could be deemed reliable. This result in far less inferred dependencies, since memory values are quite unique. Furthermore, by only allowing memory values to infer dependencies the amount of value- and def-use dependencies will be far lower than sequence-based dependencies. In general value-based dependencies are a bit more common than def-use dependencies, while sequence dependencies always exist between subsequent vertices. This fact leads to the realization that weights could be used to represent edge values. This was in part implemented in this thesis, however only by use of edge labels. The labels were not weighted, such that a def-use dependency would have different impact than other dependencies. However, by using edge labels the graphs can be compared based on structure and dependency types, rather than structure and environment specific variables such as memory values. This is important as memory values change almost every time, and should not be used for comparison between graphs.

Further analysis of the synthesize testing showed that register behavior is not traceable and thus not represented in the behavior graphs. The downside of this fact is that register behavior such as comparing register values cannot be traced. This means that comparison of variables such as if/else or string values won't be traced and cannot be part of a behavioral graph.

Differentiation in dependencies for the layers were also found during the synthesize testing. The reason for this difference is that some layers implement certain functions in different ways. This means that different layers may infer different dependencies from the same program. This in turn may lead to a difference in detection, seeing as some dependencies only may be inferred on one layer.

The experimental studies of obfuscation resilience found that 6/11 obfuscation methods had an impact on the method. This means that some methods may be ruled out completely, while the efficiency of the rest remains to be tested. The methods that could

be ruled out consists of register reassignment, white space randomization, comment randomization, variable and function name randomization as well as inlining and outlining. Register behavior is as mentioned above not captured by this method. Hence it is reasonable that it is unaffected by register-based obfuscation. White space and comment randomization are both techniques that are commonly used for JavaScripts or other code that is parsed and executed and not built by a compiler. Since the parser will ignore both whitespace and comments. The same may to some extent be applied to variable and function name randomization as well as inlining and outlining, as these are interpreted by the compiler before the executable is built. In general the method was more resilient against structural and appearance-based methods, while it was less resilient against methods that sought to change the execution flow.

Finally the method's ability to perform malware detection was analyzed by use of graph matching in SUBDUE. The results from this analysis were conflicted as the subgraphs to a certain extent was biased. The method provided good results and at best a classification accuracy of 98,9%. However due to the fact that the subgraphs used for detection was a result of compiler, it is challenging to say whether the results are valid with regard to classification accuracy.

7.2 Practical implications

The dataset used in this thesis was created based on binaries downloaded from VX heavens [127]. This is a popular dataset commonly used in research. For instance [139, 140, 41] to name a few. It is known that a lot of the malware provided at VX heavens is outdated [140], however for many purposes this dataset is sufficient. To explain how the dataset might be relevant for some and outdated for others, an understanding of features is needed. In [41] features were extracted from DLL dependencies, registry, file, network and process activity. Subsequently detection is performed at a higher layer, and the results are most likely not affected by compiler specific information. In comparison to this thesis where the library calls and system calls are traced, which as discussed in Section 6.3 are highly susceptible to changes in compilers.

This thesis found that compiler behavior such as memory protection and exception handling affected the dependency graphs. These are behaviors that were dependent on not only compiler flags, but also compiler version. As such, it is reasonable to assume that other compiler behavior might affect the dependency graphs as well. Because of this it becomes more important to be aware of differences in the dataset when choosing features. Such that relevant features are chosen. For instance, SUBDUE can be used to find 100 subgraphs. These can then be clustered and analyzed further by use of expert knowledge. A selection of subgraphs can then be performed based on subgraph behavior. For instance choosing subgraphs for botnet malware, exploits, command & control, and so on. Subsequently one can customize detection for each type of malware by choosing whether a subgraph should be included or not. This will in turn affect the detection rate and false positive rate, based on generalization of chosen subgraphs.

The complexity of graph matching was not thoroughly tested in this thesis. However, during experiments of the classifier, methods such as MDL and Size were noticeably slower than the Set cover method. Furthermore, the size of the dataset had a significant impact on the complexity and hence runtime of the experiments. Set cover and size ran for over two days on the function call dataset without being able to complete the learning

of subgraphs. This is a significant difference in comparison to Set cover with library call, which completed learning of subgraphs in the matter of a few minutes. Similarly the trace parser that builds dependencies had high runtime for large files. Both of these experiences prove as arguments against the method's scalability with regard to runtime.

7.3 Summary

The goal for this thesis was to create an automated, learning-based, obfuscation resilient detection engine able to efficiently detect malware. The motivation for this was the rapidly increasing threat of malware and computer attacks, which has led to challenges in handling security related incidents. This problem is enhanced, by the attacker's abilities to utilize unknown vulnerabilities, evasion techniques and generator algorithms, which increases the impact, effectiveness and quantity of malware. The problem has become so extensive that the typical signature-based approach no longer is sufficient. Because of this, we sought to bring detection to the lower layers of operating systems in order to increase obfuscation resilience. Furthermore the method is implemented by graph-based learning, which is able to automatically find structural patterns in data.

To achieve this, a dataset was created using an automated lab environment able to execute and an arbitrary number of malware and software. The dataset was then preprocessed to find structural patterns by use of information-based dependencies. The reliability of this method was then tested in order to make sure that no false dependencies were inferred. This was performed by experimenting with several synthesized programs, to find which variable types that could be used. The sequence of calls and order of which dependencies could be inferred was also tested. The results proved that only memory values were reliable. This is similar to the method used in [11]. However, our method can be seen as an extension to this because it utilize value- and ordering-based dependencies as well as def-use dependencies.

The method's obfuscation resilience was then tested to see whether bringing detection to a low layer increased obfuscation resilience. This turned out to be true, seeing as only 6/11 methods actually had an impact on the behavioral graphs. Furthermore the efficiency of these six methods is likely to be lower, seeing that they are all designed to obfuscate signature-based and static detection schemes.

Finally the classification accuracy of the method was to be evaluated by use of graph-based learning in SUBDUE. A best classification accuracy of 98,9% was provided and SUBDUE proved to be good at performing classification based on structural patterns. The results however, are disputed, seeing as the subgraphs used for detection was a result of compiler difference in the dataset. This lead to the realization that malware detection at this layer is far more complex, and that further precautions are needed when selecting subgraphs for detection.

8 Conclusion

This thesis has improved upon existing methodology used in [11]. Whose goal was to use dynamic traces of function call data in order to build dependencies and create graphs for malware detection. Not only has the thesis identified and analyzed key reliability issues, but it has also shown that it is resilient against a range of obfuscation techniques. The method was further expanded to use graph-based learning, by use of sub isomorphism in SUBDUE. The result is a reliable, highly resilient and an effective malware detection method. The experiments were performed based on a newly created dataset from 100 software and 200 malware, resulting in 900 traces. A total of 795 of these was chosen and used for classification and learning of subgraphs, using 10-fold cross validation.

Experimental results showed that only memory values are reliable for creating dependencies. Another crucial issue is when a dependency can be inferred. The experiments showed that hard limits with regard to dependency sequence are impossible. The use of order-based dependency was also implemented to provide more accurate behavioral graphs. Furthermore the obfuscation resilience of the method has been thoroughly tested. It is no longer a theoretical assumption based on the fact that the obfuscation techniques are designed to evade signature-based and static techniques, but a tested and verified method. The results of this analysis showed that our technique is resilient against 5/11 of the surveyed techniques. The remaining 6 techniques do to some extent impact the method, however their efficiency has not been tested.

The analysis of the method found that there are differences in performing detection at the different layers. This is not only represented by the classification accuracy, but also the fact that dependencies are a result of functions, which in turn are layer specific. The graph-based method proved to be effective in discriminating structural data. However the results may be disputed due to a difference of compilers in the dataset, which turned out to be the reason for the high classification accuracy. These results in turn lead to the realization that detection at this layer requires extraordinary attention to details regarding the selection of subgraph.

The thesis has also uncovered drawbacks of the method, which affected detection in an unexpected manner. An example of this is the fact that register behavior such as comparison of data is not exhibited in the trace files. This is important, as behavioral aspects such as if/else checks and string comparison is not a part of the behavioral graphs.

In conclusion, the thesis has made significant progress in documenting and analyzing the theoretical foundations of information-based dependency matching. The method has been deemed reliable, by extensive testing of synthesized programs. As well as resilient to a range of various obfuscation methods. The method shows great potential, but further work is needed to verify the classification accuracy and feasibility of the method with regard to computational complexity.

9 Further Work

The use of information-based dependency matching is a relatively new method, which requires further research and scrutiny in order to become more reliable. Since it is a new method there are a range of topics that should be further analyzed in order to completely understand the inherent capabilities and implications of the method. In this final chapter we discuss these issues in order to provide a pointer to topics that require further work.

First of all it is important to analyze the effects of environment specific variables such as operating system and compilers that are used to build and execute the software and malware. The effects of compilers turned out to have a significant impact on the dataset used in this thesis. It was shown that not only compilers, but compiler version, flags and options had an impact on the trace output. This should be further analyzed in order to understand the full effects this has on the dataset. The knowledge gained after analyzing effects of environment specific variables should then be used for subgraph selection. More specifically one can utilize methods such as unsupervised learning in order to cluster subgraphs based on similarity. The knowledge of environment specific variables should then be used to exclude such subgraphs from the classifier. The remaining subgraphs can be analyzed with regard to type of behavior. For instance whether the subgraph is frequent in mail communication for botnet malware, or whether it is a common buffer overflow technique typically used by exploits. By analyzing such behavior one can select subgraphs for all types of malicious behavior and extend detection coverage based on needs. Furthermore, by utilizing subgraphs for specific types of malicious behavior it is likely that the method will have better classification accuracy. More so, one can decrease the false positive rate by choosing specific and targeted behavioral subgraphs rather than generalized malicious behavior. Well-chosen subgraphs not only provide good classification accuracy, but also limit the chance of overfitting. Meaning that the method should not only perform well on this data, but provide a greater external validity and perform well on all data.

Numbering of different dependency rules was implemented for the trace parser. This resulted in the fact that dependency edges are denoted by a number, rather than the memory value. The number represents the type of dependency rule, whether it is ordering dependency, value dependency or def-use dependency. This has the beneficial effect that graph matching can be performed based on dependency types rather than environment specific values. However, the dependency rules don't occur with equal similarity. Hence it might be possible to implement weighted graph matching. Where a def-use dependency actually have a higher impact than an ordering dependency. The effects of this should be analyzed, both with regard to reliability of weighted values, as well as the resulting accuracy. Furthermore there is options in SUBDUE which allow for inexact graph matching. This means that the subgraph is not matched to identical subgraphs (isomorphism), but based on a threshold. Inexact graph matching might provide better detection rates, but is likely to also affect the false positive rate. Thus further work should be invested in analyzing whether inexact graph matching provides better classification accuracy for information-based dependency matching.

The impact of obfuscation methods was tested and results showed that 6/11 methods had an impact on the information-based dependency matching. However in order to test the efficiency of these methods more elaborate tests are needed. More specifically the dataset should include the obfuscation methods, such that tests can be performed to analyze whether this has an impact. This is challenging as a lot of malware only exist as binaries. Thus focus on source code and verification that the malware utilize the obfuscation techniques is important. This analysis was too comprehensive for this thesis, but should be analyzed as further work, as the obfuscation resilience is the key reason for choosing this detection method. Furthermore a more extensive analysis should be performed, which also includes the obfuscation methods that were not tested in this thesis due to limited time. The metrics can also be extended to include potency, resilience, stealth and cost in order to get a deeper understanding of how the obfuscation method affects information-based dependency matching.

The computational complexity of information-based dependencies and matching of the resulting graphs were not extensively tested in this thesis. Further work should be spent analyzing the complexity of the method and ultimately the possible throughput. This is important as the method cannot be used, regardless of accuracy if it is not fast enough. This relates to the psychological acceptability of the method, as users won't use it if the method is too slow.

Finally the method should be analyzed, compared and integrated with other methods. By doing this one can see how the method compares to static and signature-based approaches, as well as how good they would work together as a hybrid detection engine. Recent work has shown that combining static and dynamic detection methods provide better classification accuracy [41]. Furthermore there exist work which seek to combine different supervised and/or unsupervised classifier methods [141, 142]. Surveys comparing single and hybrid classifiers indicate that combining classifiers might improve detection rate [143]. Hence further work should be performed to survey the advantages and disadvantages of every method and analyze how the dynamic behavioral-based method proposed in this thesis can be combined with other techniques to provide a superior detection method.

Bibliography

[1] Szor, P. 2005. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional.

[2] Matt Bishop. 2003. *Computer Security Art and Science*. Addison Wesley.

[3] Yan, W., Zhang, Z., & Ansari, N. sept.-oct. 2008. Revealing packed malware. *Security Privacy, IEEE*, 6(5), 65 –69.

[4] Ben Feinstein and Daniel Peck. Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious JavaScript. `https://www.blackhat.com/presentations/bh-usa-07/Feinstein_and_Peck/Whitepaper/bh-usa-07-feinstein_and_peck-WP.pdf`. Last Visited: 14.06.2012.

[5] Jones, T. M. March 2012. Look at linux, the operating system and universal platform. `http://www.ibm.com/developerworks/library/l-linuxuniversal/`. Last Visited: 15.06.2012.

[6] Bengoetxea, E. *Inexact Graph Matching Using Estimation of Distribution Algorithms*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, France, Dec 2002.

[7] Eilam, E. & Chikofsky, E. 2005. *Reversing: secrets of reverse engineering*. Wiley.

[8] Christodorescu, M., Jha, S., Seshia, S. A., Song, D., & Bryant, R. E. 2005. Semantics-Aware Malware Detection. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, 32–46, Washington, DC, USA. IEEE.

[9] SearchSecurity. Malware definition. `http://searchmidmarketsecurity.techtarget.com/definition/malware`. Last Visited: 14.06.2012.

[10] Walker, J. August 1996. The animal episode. `http://fourmilab.ch/documents/univac/animal.html`. Last Visited: 14.06.2012.

[11] Christodorescu, M., Jha, S., & Kruegel, C. 2007. Mining specifications of malicious behavior. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, 5–14, New York, NY, USA. ACM.

[12] Sand, L. A. August 2011. Information-based dependency matching - a study of information-based dependencies of function calls. `http://www.sikkerhetsfokus.no/wp-content/uploads/2011/08/Information-based-dependency-matching-v1.pdf`. Last Visited: 14.06.2012.

[13] Preda, M. D., Christodorescu, M., Jha, S., & Debray, S. 2007. A semantics-based approach to malware detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, 377–388, New York, NY, USA. ACM.

[14] F-Secure. Boot brain virus. http://www.f-secure.com/v-descs/brain.shtml. Last Visited 14.06.2012.

[15] Yoo, I. & Ultes-Nitsche, U. 2004. How to predict e-mail viruses under uncertainty. In *Performance, Computing, and Communications, 2004 IEEE International Conference on*, 675 – 679.

[16] F-Secure. Cascade virus. http://www.f-secure.com/v-descs/cascade.shtml. Last Visited 14.06.2012.

[17] F-Secure. W32-melissa virus. http://www.f-secure.com/v-descs/melissa.shtml. Last Visited 14.06.2012.

[18] Aycock, J. 2006. *Computer Viruses and Malware*, volume 22 of *Advances in Information Security*. Springer.

[19] Staniford, S., Paxson, V., & Weaver, N. 2002. How to own the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*, 149–167, Berkeley, CA, USA. USENIX Association.

[20] Farmer, D. & Venema, W. 2004. *Forensic Discovery*. Addison Wesley Professional.

[21] Kolishak, A. December 2003. Toctou with nt system service hooking. http://seclists.org/bugtraq/2003/Dec/351. Last Visited 14.06.2012.

[22] matousec. May 2010. Khobe - 8 earthquake for windows desktop security software. http://www.matousec.com/info/articles/khobe-8.0-earthquake-for-windows-desktop-security-software.php. Last Visited 14.06.2012.

[23] Bishop, M. & Dilger, M. 1996. Checking for race conditions in file accesses. http://nob.cs.ucdavis.edu/bishop/papers/1996-compsys/racecond.pdf. Last Visited: 14.06.2012.

[24] Sparks, S. & Butler, J. August 2005. Shadow Walker: Raising The Bar For Windows Rootkit Detection. *Phrack*, 11(63).

[25] Tellefsen, A. 2010. Harware trojan horses - taxonomy and defense techniques. 27.

[26] Kalafut, A., Acharya, A., & Gupta, M. 2006. A study of malware in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, 327–332, New York, NY, USA. ACM.

[27] Scarfone, K. & Mell, P. February 2007. Guide to intrusion detection and prevention systems - recommendations of the national institute of standards and technology. http://csrc.nist.gov/publications/nistpubs/800-94/SP800-94.pdf. SP 800-94, Last Visited: 14.06.2012.

[28] Petrovic, S. Intrusion detection and prevention lecture 1 - definition and classification. Lecture Material from fronter.com/hig.

[29] Idika, N. & Mathur, A. P. 2007. A survey of malware detection techniques. *Purdue University*, 48.

[30] Christodorescu, M. & Jha, S. 2003. Static analysis of executables to detect malicious patterns. In *In Proceedings of the 12th USENIX Security Symposium*, 169–186.

[31] Project, T. O. W. A. S. November 2011. Source code analysis tools. `https://www.owasp.org/index.php/Source_Code_Analysis_Tools`. Last Visited: 14.06.2012.

[32] Ligh, M., Ligh, M., Adair, S., Richard, M., & Hartstein, B. 2010. *Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code*. John Wiley & Sons.

[33] Ormand, T. An empirical study into the security exposure to hosts of hostile virtualized environments. `http://taviso.decsystem.org/virtsec.pdf`. Last Visited: 14.06.2012.

[34] Zeltser, L. Introduction to malware analysis - free recorded webcast. `http://zeltser.com/reverse-malware/malware-analysis-webcast.html`. Last Visited: 14.06.2012.

[35] Regshot. `http://code.google.com/p/regshot/`. Last Visited: 14.06.2012.

[36] Windows. Sysinternals. `http://technet.microsoft.com/en-us/sysinternals/bb545021`. Last Visited: 14.06.2012.

[37] Ferguson, J. & Kaminsky, D. 2008. *Reverse engineering code with IDA Pro*. Syngress Pub.

[38] Arini BalaKrishnan, C. S. December 2005. Code obfuscation literature survey. `http://pages.cs.wisc.edu/~arinib/writeup.pdf`. Last Visited: 14.06.2012.

[39] Collberg, C., Thomborson, C., & Low, D. 1997. A taxonomy of obfuscating transformations. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.9852`. Last Visited: 14.06.2012.

[40] You, I. & Yim, K. 2010. Malware obfuscation techniques: A brief survey. In *Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, BWCCA '10, 297–300, Washington, DC, USA. IEEE Computer Society.

[41] Berg, P. E. Bahavior-based classification of botnet malware. Master's thesis, Gjovik University College, 2010.

[42] Markus Franz Xaver Johannes Oberhumer, Laszlo Molnarberhumer, J. F. R. Ultimate packer for executables. `http://upx.sourceforge.net/`. Last Visited: 14.06.2012.

[43] Softidentity. Aspack. `www.aspack.com`. Last Visited: 14.06.2012.

[44] Unknown. Yoda's crypter. `http://yodas-crypter.softpedia.com/`. Last Visited: 14.06.2012.

[45] JlabSoftware. Polycrypt pe. http://download.cnet.com/PolyCrypt-PE/3000-2092_4-10401299.html. Last Visited: 14.06.2012.

[46] Toolworks, T. S. R. Armadillo. http://www.siliconrealms.com/. Last Visited: 14.06.2012.

[47] Technologies, O. Themida. http://www.oreans.com/themida.php. Last Visited: 14.06.2012.

[48] Bitsum. Pebundle. http://www.bitsum.com/pebundle.asp. Last Visited: 14.06.2012.

[49] Software, T. Molebox. http://www.molebox.com/. Last Visited: 14.06.2012.

[50] Oberheide, J., Bailey, M., & Jahanian, F. August 2009. PolyPack: An Automated Online Packing Service for Optimal Antivirus Evasion. In *3rd USENIX Workshop on Offensive Technologies (WOOT'09)*, Montreal, Canada.

[51] Schiffman, M. February 2010. A brief history of malware obfuscation part 2. http://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_2_of_2/. Last Visited: 14.06.2012.

[52] Karthik Selvaraj and Nino Fred Gutierrez. The Rise of PDF Malware. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_rise_of_pdf_malware.pdf. Last Visited: 14.06.2012.

[53] Tanenbaum, A. S. 2007. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition.

[54] Reddy, D. & Pujari, A. 2006. N-gram analysis for computer virus detection. *Journal in Computer Virology*, 2, 231–239. 10.1007/s11416-006-0027-8.

[55] Firdausi, I., Lim, C., Erwin, A., & Nugroho, A. dec. 2010. Analysis of machine learning techniques used in behavior-based malware detection. In *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*, 201 –203.

[56] Yu, S., Zhou, S., Liu, L., Yang, R., & Luo, J. 2010. Malware variants identification based on byte frequency. In *Proceedings of the 2010 Second International Conference on Networks Security, Wireless Communications and Trusted Computing - Volume 02*, NSWCTC '10, 32–35, Washington, DC, USA. IEEE Computer Society.

[57] Li, W.-J., Wang, K., Stolfo, S., & Herzog, B. june 2005. Fileprints: identifying file types by n-gram analysis. In *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*, 64 – 71.

[58] Saxena, P., Sekar, R., & Puranik, V. 2008. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, 74–83, New York, NY, USA. ACM.

[59] Avancini, A. & Ceccato, M. 2010. Towards security testing with taint analysis and genetic algorithms. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, SESS '10, 65–71, New York, NY, USA. ACM.

[60] Smith, G. 2007. Principles of secure information flow analysis. In *Malware Detection*, 297–307. Springer-Verlag.

[61] Zhang, Y., Pang, J., Yue, F., & Cui, J. 2010. Fuzzy neural network for malware detect. *Intelligent System Design and Engineering Application, International Conference on*, 1, 780–783.

[62] Sami, A., Yadegari, B., Rahimi, H., Peiravian, N., Hashemi, S., & Hamze, A. 2010. Malware detection based on mining api calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, 1020–1025, New York, NY, USA. ACM.

[63] Lee, J., Jeong, K., & Lee, H. 2010. Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, 1970–1977, New York, NY, USA. ACM.

[64] Christodorescu, M. & Jha, S. 2003. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, 12–12, Berkeley, CA, USA. USENIX Association.

[65] Kruegel, C., Robertson, W., & Vigna, G. 2004. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, 91–100, Washington, DC, USA. IEEE Computer Society.

[66] Kruegel, C. 2005. Behavioral and structural properties of malicious code.

[67] Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H., & Munchen, T. U. 2005. Detecting malicious code by model checking. In *Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA 05), volume 3548 of Lecture Notes in Computer Science*, 174–187. Springer Berlin.

[68] Guo, H., Pang, J., Zhang, Y., Yue, F., & Zhao, R. oct. 2010. Hero: A novel malware detection framework based on binary translation. In *Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE International Conference on*, volume 1, 411 –415.

[69] Ye, Y., Wang, D., Li, T., & Ye, D. 2007. Imds: intelligent malware detection system. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, 1043–1047, New York, NY, USA. ACM.

[70] Jonathan-Christofer Demay, Frederic Majorczyk, E. T. & Tronel, F. 2011. Detecting illegal system calls using a data-oriented detection model. In *26th International Information Security Conference (SEC 2011) June*.

[71] Shankarapani, M., Ramamoorthy, S., Movva, R., & Mukkamala, S. 2011. Malware detection using assembly and api call sequences. *Journal in Computer Virology*, 7, 107–119. 10.1007/s11416-010-0141-5.

[72] Chandola, V., Boriah, S., & Kumar, V. 2010. A reference based analysis framework for analyzing system call traces. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, 33:1–33:3, New York, NY, USA. ACM.

[73] Park, Y., Reeves, D., Mulukutla, V., & Sundaravel, B. 2010. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW '10, 45:1–45:4, New York, NY, USA. ACM.

[74] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X., & Wang, X. 2009. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, 351–366, Berkeley, CA, USA. USENIX Association.

[75] Pu, S. & Lang, B. 2007. An intrusion detection method based on system call temporal serial analysis. In *Proceedings of the intelligent computing 3rd international conference on Advanced intelligent computing theories and applications*, ICIC'07, 656–666, Berlin, Heidelberg. Springer-Verlag.

[76] Yap, H. C. 2006. Improving host-based ids with argument abstraction to prevent mimicry attacks. *RAID 2005, LNCS*, 3858, 146–164.

[77] Mehdi, S. B., Tanwani, A. K., & Farooq, M. 2009. Imad: in-execution malware analysis and detection. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, GECCO '09, 1553–1560, New York, NY, USA. ACM.

[78] Tokhtabayev, A. G., Skormin, V. A., & Dolgikh, A. M. 2010. Expressive, efficient and obfuscation resilient behavior based ids. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, 698–715, Berlin, Heidelberg. Springer-Verlag.

[79] Jyostna, G., Himanshu, P., & Eswari, P. R. L. 2011. Detecting anomalous application behaviors using a system call clustering method over critical resources. In *Advances in Network Security and Applications*, Wyld, D. C., Wozniak, M., Chaki, N., Meghanathan, N., & Nagamalai, D., eds, volume 196 of *Communications in Computer and Information Science*, 53–64. Springer Berlin Heidelberg. 10.1007/978-3-642-22540-6_6.

[80] Wee, K. & Moon, B. 2003. Automatic generation of finite state automata for detecting intrusions using system call sequences. In *Computer Network Security*, Gorodetsky, V., Popyack, L., & Skormin, V., eds, volume 2776 of *Lecture Notes in Computer Science*, 206–216. Springer Berlin / Heidelberg. 10.1007/978-3-540-45215-7_17.

[81] Xu, X. & Xie, T. 2005. A reinforcement learning approach for host-based intrusion detection using sequences of system calls. In *Advances in Intelligent Computing*, Huang, D.-S., Zhang, X.-P., & Huang, G.-B., eds, volume 3644 of *Lecture Notes in Computer Science*, 995–1003. Springer Berlin / Heidelberg. 10.1007/11538059_103.

[82] Wang, W., Guan, X., & Zhang, X. 2004. A novel intrusion detection method based on principal component analysis. In *in Computer Security, Advances in Neural Networks, International IEEE Symposium on Neural Networks*, 657–662. Lecture.

[83] Guo, S., Yuan, Q., Lin, F., Wang, F., & Ban, T. 2010. A malware detection algorithm based on multi-view fusion. In *Proceedings of the 17th international conference on Neural information processing: models and applications - Volume Part II*, ICONIP'10, 259–266, Berlin, Heidelberg. Springer-Verlag.

[84] Ahmed, F., Hameed, H., Shafiq, M. Z., & Farooq, M. 2009. Using spatio-temporal information in api calls with machine learning algorithms for malware detection. In *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*, AISec '09, 55–62, New York, NY, USA. ACM.

[85] Frossi, A., Maggi, F., Rizzo, G. L., & Zanero, S. 2009. Selecting and improving system call models for anomaly detection. In *Proceedings of the 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '09, 206–223, Berlin, Heidelberg. Springer-Verlag.

[86] Qian, Q. & Xin, M. 2007. Research on hidden markov model for system call anomaly detection. In *Proceedings of the 2007 Pacific Asia conference on Intelligence and security informatics*, PAISI'07, 152–159, Berlin, Heidelberg. Springer-Verlag.

[87] Ho, A., Fetterman, M., Clark, C., Warfield, A., & Hand, S. 2006. Practical taint-based protection using demand emulation. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, 29–41, New York, NY, USA. ACM.

[88] Yin, H., Song, D., Egele, M., Kruegel, C., & Kirda, E. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, 116–127, New York, NY, USA. ACM.

[89] Zhang, X.-S., Zhi, L., & Chen, D.-P. 2008. A practical taint-based malware detection. *2008 International Conference on Apperceiving Computing and Intelligence Analysis*, 73–77.

[90] Kong, J., Zou, C. C., & Zhou, H. 2006. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, 18–24, New York, NY, USA. ACM.

[91] Maggi, F., Matteucci, M., & Zanero, S. oct.-dec. 2010. Detecting intrusions through system call sequence and argument analysis. *Dependable and Secure Computing, IEEE Transactions on*, 7(4), 381 –395.

[92] Mutz, D., Valeur, F., Vigna, G., & Kruegel, C. February 2006. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9, 61–93.

[93] Kruegel, C., Mutz, D., Valeur, F., & Vigna, G. 2003. On the detection of anomalous system call arguments.

[94] P., V., Jain, H., Golecha, Y. K., Gaur, M. S., & Laxmi, V. 2010. Medusa: Metamorphic malware dynamic analysis usingsignature from api. In *Proceedings of the 3rd international conference on Security of information and networks*, SIN '10, 263–269, New York, NY, USA. ACM.

[95] Dinaburg, A., Royal, P., Sharif, M., & Lee, W. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, 51–62, New York, NY, USA. ACM.

[96] Mutz, D., Robertson, W., Vigna, G., & Kemmerer, R. 2007. Exploiting execution context for the detection of anomalous system calls. In *Proceedings of the 10th international conference on Recent advances in intrusion detection*, RAID'07, 1–20, Berlin, Heidelberg. Springer-Verlag.

[97] Zhang, X., Li, J., Jiang, Z., & Feng, H. 2007. Black-box extraction of functional structures from system call traces for intrusion detection. In *ICIC (3)*, 135–144.

[98] Battistoni, R., Gabrielli, E., & Mancini, L. 2004. A host intrusion prevention system for windows operating systems. In *Computer Security ESORICS 2004*, Samarati, P., Ryan, P., Gollmann, D., & Molva, R., eds, volume 3193 of *Lecture Notes in Computer Science*, 352–368. Springer Berlin / Heidelberg. 10.1007/978-3-540-30108-0_22.

[99] Bernaschi, M., Gabrielli, E., & Mancini, L. V. February 2002. Remus: a security-enhanced operating system. *ACM Trans. Inf. Syst. Secur.*, 5, 36–61.

[100] Moser, A., Kruegel, C., & Kirda, E. dec. 2007. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, 421 –430.

[101] Jaeger, T. Reference monitor. `http://www.cse.psu.edu/~tjaeger/cse544-s10/papers/refmon.pdf`. Systems and Internet Infrastructure Security Lab Pennsylvania State University, Last Visited: 14.06.2012.

[102] International Secure Systems Lab, V. U. o. T. Anubis: Analyzing unknown binaries. `http://anubis.iseclab.org/`. Last Visited: 14.06.2012.

[103] Gao, X., Xiao, B., Tao, D., & Li, X. 2010. A survey of graph edit distance. *Pattern Analysis & Applications*, 13, 113–129. 10.1007/s10044-008-0141-y.

[104] Bai, L., Pang, J., Zhang, Y., Fu, W., & Zhu, J. 2009. Detecting malicious behavior using critical api-calling graph matching. In *Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering*, ICISE '09, 1716–1719, Washington, DC, USA. IEEE Computer Society.

[105] Garey, M. R. & Johnson, D. S. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

[106] Ammar Ahmed E. Elhadi, Mhd Aizaini Maarof, A. H. O. Malware detection based on hybrid signature behavior application programming interface call graph. http://dx.doi.org/10.3844/ajassp.2012.283.288.

[107] Abdulrahim, M. A. *Parallel algorithms for labeled graph matching*. PhD thesis, Golden, CO, USA, 1998. AAI0599838.

[108] Theodoridis, S. & Koutroumbas, K. 2008. *Pattern Recognition, Fourth Edition*. Academic Press, 4th edition.

[109] Holder, L. 2004. Graph-based learning. `http://eecs.wsu.edu/~holder/courses/cse6363/spr04/slides/GBL.pdf`. Lecture notes, Last Visited 14.06.2012.

[110] Diane Cook, L. H. Subdue graph based knowledge discovery. `http://ailab.wsu.edu/subdue/`. Last visited: 18.05.2012.

[111] Noble, C. C. & Cook, D. J. 2003. Graph-based anomaly detection. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, 631–636, New York, NY, USA. ACM.

[112] The SUBDUE project. Subdue manual. `http://ailab.wsu.edu/subdue/`. Last visited: 21.05.2012.

[113] DARPA Intrusion Detection Evaluation Group. 1998. 1998 darpa intrusion detection evaluation data set. `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/1998data.html`. Last Visited 14.06.2012.

[114] DARPA Intrusion Detection Evaluation Group. 1999. 1999 darpa intrusion detection evaluation data set. `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/1999data.html`. Last Visited 14.06.2012.

[115] DARPA Intrusion Detection Evaluation Group. 2000. 2000 darpa intrusion detection scenario specific data sets. `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/2000data.html`. Last Visited 14.06.2012.

[116] University of New Mexico. Sequence time-delay embedding (stide) benchmark data sets. `http://www.cs.unm.edu/~immsec/systemcalls.htm`. Last Visited 14.06.2012.

[117] VMware. Vmware. `http://www.vmware.com/`. Last Visited 14.06.2012.

[118] Oracle. Virtualbox. `https://www.virtualbox.org/`. Last Visited 14.06.2012.

[119] Zeltser, L. August 2011. Vmware network isolation for malware analysis lab. `http://blog.zeltser.com/post/8978449246/vmware-network-isolation-for-malware-analysis`. Last Visited: 14.06.2012.

[120] Zimmer, D. Malcode analysis pack. `labs.idefense.com/files/labs/releases/previews/map`. 10.01.2012.

[121] Bellard, F. Qemu - open source processor emulator. `http://wiki.qemu.org/`. Last Visited: 14.06.2012.

[122] Koret, J. Zero wine - malware behavior analysis. `http://zerowine.sourceforge.net/`. Last Visited: 14.06.2012.

[123] Bin, C. J. Zero win tryouts. http://zerowine-tryout.sourceforge.net/. Last Visited: 14.06.2012.

[124] Garg, P. Stracent - system call tracer for windows nt. http://www.intellectualheaven.com/Articles/StraceNT.pdf. Last Visited: 14.06.2012.

[125] Kaplan, Y. Api spying techniques for windows 9x nt and 2000. http://www.internals.com/articles/apispy/apispy.htm. Last Visited: 14.06.2012.

[126] Wright, J. March 2003. Remote kernel debugging with windbg. http://www.wd-3.com/archive/RemoteDbg.htm. Last Visited: 14.06.2012.

[127] Vx heavens. vx.netlux.org. Last Visited: 15.03.2012.

[128] Packet storm security. packetstormsecurity.org. Last Visited: 14.06.2012.

[129] Offensive computing. http://www.offensivecomputing.net/. Last Visited: 15.03.2012.

[130] Witten, I. H., Frank, E., & Hall, M. A. 2011. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, Amsterdam, 3. edition.

[131] Mark Hall, Eibe Frank, G. H. B. P. P. R. I. H. W. 2009. The weka data mining software: An update. http://www.cs.waikato.ac.nz/ml/weka/. SIGKDD Explorations, Volume 11, Issue 1.

[132] Minka, T. A statistical learning/pattern recognition glossary. http://alumni.media.mit.edu/~tpminka/statlearn/glossary/. Last visited: 26.05.2012.

[133] Payne, S. C. June 2006. A Guide to Security Metrics. http://www.sans.org/reading_room/whitepapers/auditing/guide-security-metrics_55. Last Visited: 14.06.2012.

[134] Kerrisk, M. Linux kernel access. http://www.kernel.org/doc/man-pages/online/pages/man2/access.2.html. Last Visited: 14.06.2012.

[135] Kerrisk, M. Linux kernel mmap. http://www.kernel.org/doc/man-pages/online/pages/man2/mmap.2.html. Last Visited: 14.06.2012.

[136] Keating, G. Register frame info problem on alpha. http://gcc.gnu.org/ml/gcc/1999-05n/msg00755.html. Last Visited: 14.06.2012.

[137] Schulze, M. Register frame info. http://www.infodrom.north.de/~joey/Linux/Tips+Tricks/register-frame.html.en.html. Last Visited 20.03.2012.

[138] Security, O. Exploit-db. http://www.exploit-db.com/. Last Visited: 14.06.2012.

[139] Nataraj, L., Yegneswaran, V., Porras, P., & Zhang, J. 2011. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, AISec '11, 21–30, New York, NY, USA. ACM.

[140] Laskov, P. & Šrndić, N. 2011. Static detection of malicious javascript-bearing pdf documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, 373–382, New York, NY, USA. ACM.

[141] Z. Muda, W. Yassin, M. S. & Udzir, N. 2011. A k-means and naive bayes learning approach for better intrusion detection. In *Information Technology Journal 10*, volume 10.

[142] Kim, D. S., Nguyen, H.-N., Ohn, S.-Y., & Park, J. S. 2005. Fusions of ga and svm for anomaly detection in intrusion detection system. In *ISNN (3)*, 415–420.

[143] Sand, L. A. 2011. Network intrusion detection - a survey of supervised classification techniques. `http://www.sikkerhetsfokus.no/wp-content/uploads/2012/06/Network-Intrusion-Detection-A-Survey-of-Supervised-Classification-Techniques-v2.pdf`. Last Visited: 14.06.2012.

A Output parser

This appendix consist of the output parser. The output parser was used for pre-processing trace data and create structural behavior based on information-based dependency matching. The parser include all three types of dependency rules and can parse logs from strace and ltrace for function calls, library calls and system calls.

```
#include "listtool2.h"
#include <iostream>
#include <fstream>
#include <stdio.h>
#include <string.h>
#include <string>
#include <algorithm>
#include <ctime>

using namespace std;

    //CONSTANTS
const int STRLEN=500;   //Constant length of strings
bool type; // 0=Malware, 1=Software
int layer; // 0=Library, 1= System, 3=Hybrid/Function

    //GLOBAL VARIABLES
List*  Function_calls;  //List containing function calls
int func_calls_nr=0;    //Nr of elements in list

    //CLASSES
class Function_call : public Num_element {
 private:
  double rtime;     //Execution time of function call
  double continued; //Whether the function call is divided in input file or not
  char* name;       //Name of function call
  List* Parameters; //List of parameters
  List* Dependencies;   //List of dependencies
  char* ret_value;  //Return calue of function clal
  int nr;           //Number of parameters
 public:
  Function_call(int t, char str[STRLEN], double time); //Constructor
  bool sim_check(char str[STRLEN]); //Similarity checker
  void constructor_continued(char str[STRLEN], double cont);    //Constructor for unfinnished function calls
  bool argument_comparison(char* returnvalue); //Compare return value
  char* ret_retvalue(); //Return return value
  char* ret_name(); //return name
  double ret_rtime(); //return execution time
  void add_dependency(int i, char* d_name, int d_nr, int t, double d_rtime, char* edge);    //add dependency
  bool dep_check(); //check dependency
  void write_name(ofstream* out);   //write name to file
  void writesubdue_name(ofstream* out); //write name to subdue file
  void write_dep(ofstream* out);    //write dependencies to file
  void writesubdue_dep(ofstream* out); //write vertice/edges to subdue file
  int retparnr();   //return parameter nr
  char* retparstring(int i); //return paremeter string
};

class Dependency : public Num_element {
private:
    char* name;     //name of dependency node
    int d_nr; //number of dependency node
    int type;   //1=ordering, 2=value 3=def-use
```

```
    double rtime;   //timestamp of dependency node
    char* ret_value; //value of dependency (edge)
public:
    Dependency(int i, char* dep_name, int nr, int t, double dep_rtime, char* dep_retval); //constructor
    void write(ofstream* out);  //write dependency to file
    void writesubdue(ofstream* out); //write to subdue file
    char *retval();
};

class Parameter : public Num_element {
 private:
  int argnr;        //Argument number
  char* arg_name;   //argument name
 public:
  Parameter(int nr, char str[STRLEN]);  //constructor
  bool comparison(char* check); //Compare parameter name
  char* retval();
};

    //GLOBAL FUNCTIONS
void stringparser(char str[STRLEN], bool exception);
char stringextracter(char in[STRLEN], char out[STRLEN],int* counter, bool var);
void readfile(char* filename);
void fixstring(char in[STRLEN], char out[STRLEN], int from, int size);
void create_dependencies();
void write_dependencies(char* filename);
void write_subdue(char* filename);
bool valuedep(Function_call* call, int nr, Function_call* call_comp);
bool handlechecker(char* str);
void checktype(char* filename);

int main(int argc, char **argv) {
    clock_t start;  //clocke for performance measurement
    double duration;
    start = clock(); //Performance measuring start time


    Function_calls = new List(Sorted); //Create list

    readfile(argv[1]);  //reads the file with argument 1 as file name
    cout << "\nread complete";
    checktype(argv[1]);
    cout << "\nchecktype complete";
    create_dependencies();  //creates dependencies
    cout << "\ndependencies complete";
    write_dependencies(argv[2]);    //writes file with argument 2 as file name
    cout << "\ndot-file complete";
    write_subdue(argv[3]);  //write vertice/edges to subdue file
    cout << "\nsubdue-file complete";

    duration = ( clock() - start) / (double) CLOCKS_PER_SEC; //stop clock
    cout << "\n\nNUMER OF FUNCTION CALLS: " << func_calls_nr; //print call nr
    cout << "\nRUNTIME: " << duration << " seconds\n\n";  //print duration
```

```
107
108     return 0;   //exit
109 }
110     //GLOBAL FUNCTIONS DEFINED
111 void write_dependencies(char* filename) {
112     Function_call* call;    //function_call object
113
114     ofstream out(filename); //create file
115     out << "digraph G {\n";   //write metadata for dot.exe
116     for(int i=1;i<=func_calls_nr;i++) {  //parse and write all nodes
117         call = (Function_call*) Function_calls->remove_no(i);
118         call->write_name(&out);
119         Function_calls->add(call);
120     }
121     for(int i=1;i<=func_calls_nr;i++) {  //parse and write all edges
122         call = (Function_call*) Function_calls->remove_no(i);
123         if(call->dep_check()==true)
124             call->write_dep(&out);
125         Function_calls->add(call);
126     }
127     out << "}\n";
128 }
129
130 void write_subdue(char* filename) { //append vertice/edges to subdue files
131     Function_call* call; //function call object
132
133     ofstream out;
134     if(layer==0) //if library call
135         out.open("library.g",ios::app);
136
137     if(layer==1) //if kernel call
138         out.open("kernel.g",ios::app);
139
140     if(layer==2) //if function call
141         out.open("function.g",ios::app);
142
143     out << "% " << filename << "\n";
144     if(type==0) //malware
145         out << "XP\n";
146     else //software
147         out << "XN\n";
148     for(int i=1;i<=func_calls_nr;i++) {
149         out << "v " << i << " ";
150         call = (Function_call*) Function_calls->remove_no(i);
151         call->writesubdue_name(&out);
152         Function_calls->add(call);
153     }
154     for(int i=1;i<=func_calls_nr;i++) {
155         call = (Function_call*) Function_calls->remove_no(i);
156         if(call->dep_check()==true)
157             call->writesubdue_dep(&out);
158         Function_calls->add(call);
159     }
160 }
```

```
161
162 void create_dependencies() {    //This function parse and create dependencies based on return value and parameters
163     Function_call* call;        //function_call object
164     Function_call* call_comp;   //x2
165
166     //ordering-based dependencies
167     for(int i=1;i<=func_calls_nr;i++) {  //Parse all function calls
168         call = (Function_call*) Function_calls->remove_no(i);
169         if(i>1) {
170             call_comp = (Function_call*) Function_calls->remove_no(i-1);
171             call_comp->add_dependency(i,call->ret_name(),i,1 ,call->ret_rtime(),"1");
172             Function_calls->add(call_comp);
173             }
174         Function_calls->add(call);
175     }
176
177     //value-based dependencies
178     for(int i=1;i<=func_calls_nr;i++) {  //Parse all function calls
179         call = (Function_call*) Function_calls->remove_no(i);
180         for(int j=i-1;j>0;j--) { //Parse all function calls which are smaller than current to check for dependencies
181             call_comp = (Function_call*) Function_calls->remove_no(j);
182             if(valuedep(call, i, call_comp)==true) {
183                 Function_calls->add(call_comp);
184                 break;  //Dependency added and function breaks as soon as dependency is found. I.e. only 1
185             }           //possible dependency is extracted
186             Function_calls->add(call_comp);
187         }
188         Function_calls->add(call);
189     }
190
191     //def-use dependencies
192     for(int i=1;i<=func_calls_nr;i++) {  //Parse all function calls
193         call = (Function_call*) Function_calls->remove_no(i);
194         for(int j=i-1;j>0;j--) { //Parse all function calls which are smaller than current to check for dependencies
195             call_comp = (Function_call*) Function_calls->remove_no(j);
196             if(handlechecker(call_comp->ret_retvalue())==true && call->argument_comparison(call_comp->ret_retvalue())==true) {
197                 call_comp->add_dependency(i,call->ret_name(),i,3,call->ret_rtime(),NULL);
198                 Function_calls->add(call_comp);
199                 break;  //Dependency added and function breaks as soon as dependency is found. I.e. only 1
200             }           //possible dependency is extracted
201             Function_calls->add(call_comp);
202         }
203         Function_calls->add(call);
204     }
205
206 }
207
208 bool valuedep(Function_call* call, int nr, Function_call* call_comp) {
209     int i,j, cnr, cmpnr;    //function check whether a value-based dependency exist
210     char *cstring;
211     char *cmpstring;
212     cnr=call->retparnr();
213     cmpnr=call_comp->retparnr();
```

```
214
215         for(i=1;i<=cnr;i++) {
216             cstring = call->retparstring(i);
217             for(j=1;j<=cmpnr;j++) {
218                 cmpstring = call_comp->retparstring(j);
219                 if(!strcmp(cstring,cmpstring) && handlechecker(cstring)==true) {
220                     call_comp->add_dependency(nr,call->ret_name(),nr,2,call->ret_rtime(),cstring);
221                     return true;
222                 }
223             }
224         }
225         return false;
226     }
227
228
229     void stringparser(char str[STRLEN], bool exception) {
230         Function_call* new_call;    //function call object
231         int total=0;                //temp variable
232         double t,m,s,mc, rtime;     //temp time variables
233         char time[STRLEN],tt[STRLEN],mm[STRLEN],ss[STRLEN],mcmc[STRLEN]; //temp time variables
234
235         fixstring(str,time,0,15);   //the following lines parse timestamp string
236         fixstring(time,tt,0,2);     //then change format such that time
237         fixstring(time,mm,3,2);     //fit in a double value
238         fixstring(time,ss,6,2);
239         fixstring(time,mcmc,9,6);
240         t=atof(tt);
241         m=atof(mm);
242         s=atof(ss);
243         mc=atof(mcmc);
244         rtime=(t*3600)+(m*60)+s+(mc/1000000);
245
246         if(exception==false) {      //if new function call
247             new_call = new Function_call(++func_calls_nr,str,rtime);
248             Function_calls->add(new_call);
249         }
250         else {  //if interrupted function call
251             total=Function_calls->no_of_elements();
252             for(int i=1;i<=total;i++) {
253                 new_call = (Function_call*) Function_calls->remove_no(i);
254                 if(new_call->sim_check(str)==true) //if found
255                     new_call->constructor_continued(str,rtime);  //continue constructor
256                 Function_calls->add(new_call);
257             }
258         }
259     }
260
261     void fixstring(char in[STRLEN], char out[STRLEN], int from, int size)  {
262         int max,j;      //Extract a string from a string based on
263         max=from+size;  //length of string and initial value
264         j=0;
265         for(int i=from; i<max; i++) {
266             out[j]=in[i];
267             j++;
```

```
    }
    out[j]='\0';
}

void checktype(char* filename) { //check type of string
    if(strstr(filename,"Backdoor")!=0 || strstr(filename,"DoS")!=0 || strstr(filename,"Exploit")!=0 || strstr(filename,"Flooder")!=0
        type=0; //is malware
    else
        type=1; //is software

    if(strstr(filename,"library_call")!=0)
        layer=0;
    else if(strstr(filename,"system_call")!=0)
        layer=1;
    else if(strstr(filename,"function_call")!=0)
        layer=2;
}

void readfile(char* filename) { //Function for parsing file
    /*HAD AN ISSUE WITH CORRUPTED POINTERS BY USING REGULAR
    GETLINE/GET FUNCTIONS FROM THESE FILES. THEREFORE A
    CHARACTER BY CHARACTER EXTRACTION IS USED*/
    char dummy[STRLEN]; //temp string
    int counter=0;      //temp variable
    bool eos1=false;    //control flow temp variables
    bool eos2=false;
    bool endoffile=false;

    ifstream infile(filename);  //open file
    if(infile) {                //if open successful
        cout << "\nReading from: " << filename << endl;

        while(endoffile==false) {   //reads until end of file
            counter=0;              //initialize array value
            dummy[counter]=' ';
            endoffile=infile.eof();
            while(dummy[counter-1]!='\n') { //read until end of line
                dummy[counter] = infile.get();
                dummy[++counter] = '\0';
                if(infile.eof()) {
                    endoffile=true;
                    break;
                }
            }
            if(strstr(dummy,"+++ exited")!=0 || strstr(dummy,"exit_group")!=0 || endoffile) { //end of file found
                endoffile=true;
                break;
            }
            if(strstr(dummy,"unfinished ...>")==0 && strstr(dummy," = ")==0) { //if not complete string
                break;
            }
            if(strstr(dummy,"fgets resumed>")!=0) //MUST BE REMOVED
                cout << "here";
```

```
            if(strstr(dummy,"resumed>")!=0) //read until resumed
                stringparser(dummy,true); //fractioned function call found
            else
                stringparser(dummy,false); //new function call found
        }
    }
    else {
        cout << "\nCould not find: " << filename << "...\n";
    }
}

char stringextracter(char in[STRLEN], char out[STRLEN],int* counter, bool var) {
    int nr=0;
    char ret;
    bool foundret=0;
    bool exception=false;

    out[nr++] = in[(*counter)++];
    out[nr] = '\0'; //read until the following distinguished characters are found (metadata characters)
//  while(out[nr-1]!='(' && out[nr-1]!=','&& out[nr-1]!='=' && out[nr-1]!=')' && strstr(out,"<unfinished ...>")==0) {
    while(exception==false && out[nr-1]!=','&& foundret==false && out[nr-1]!=')' && strstr(out,"<unfinished ...>")==0) {
        exception=false;
        out[nr++] = in[(*counter)++];
        out[nr] = '\0';
        if(out[nr-1]=='(' && var==false) //exc handling //only allow one name
            exception=true;
        if(strstr(out," =")) {
            foundret=true;
        }
    }
    ret=out[nr-1];
    out[nr-1]='\0';
    return ret; //then return the string up untill that character
}

    //CLASS FUNCTIONS DEFINED
char* Function_call::ret_name() {
    return name;    //return function call name
}

double Function_call::ret_rtime() {
    return rtime;   //return function call time
 }

Function_call :: Function_call(int t, char str[STRLEN], double time) : Num_element(t) {
    char dummy[STRLEN]; //This call takes the string from readfile as input and crates a function call
    int counter=16;     //based on the string
    char pathfinder;
    Parameter* new_par;
    bool jump=false;
    int n, c;
    nr=0;
    rtime=time; //rtime is set
    bool var=false; //find whether name has been found and var is next (exception handler)
```

```
375
376         Dependencies = new List(Sorted);    //lists created
377         Parameters = new List(Sorted);
378
379         continued=0;                    //continued zeroed
380         ret_value= new char[8];         //ret value set such that unfinnished function calls can be detected later on
381         strcpy(ret_value,"NOT_SET");
382
383         pathfinder=stringextracter(str,dummy,&counter,var); //find metadata characters
384
385         while(pathfinder!='=' && pathfinder!='.') { //while not end of file or fragmented file
386
387             if(pathfinder=='(') {   //file name string
388                 name= new char[strlen(dummy)+1];
389                 replace(dummy,dummy+strlen(dummy),'=','_');
390                 strcpy(name,dummy);
391                 var=true;
392             }
393             else if(pathfinder==',') { //parameter string
394                 new_par = new Parameter(++nr,dummy);
395                 Parameters->add(new_par);
396                 counter+=1;
397             }
398             else if(pathfinder=='>') { //fragmented string
399                 dummy[strlen(dummy)-16]='\0';
400                 new_par= new Parameter(++nr,dummy);
401                 Parameters->add(new_par);
402                 jump=true;
403                 break;
404             }
405             else if(pathfinder==')') { //last parameter string
406                 if(strlen(dummy)>0) {
407                     new_par = new Parameter(++nr,dummy);
408                     Parameters->add(new_par);
409                 }
410             }
411             pathfinder=stringextracter(str,dummy,&counter,var); //continue reading string
412         }
413         if(jump==false) {   //if not fragmented file
414             n=0;            //then read untill end of file
415             counter++;      //and extract the return value
416             dummy[n++]=str[counter++];
417             dummy[n]='\0';
418
419             while(dummy[strlen(dummy)-1]!='\n' && dummy[strlen(dummy)-1]!=' ') {
420                 dummy[n++]=str[counter++];
421                 dummy[n]='\0';
422             }
423             dummy[--n]='\0';
424             ret_value = new char[n];
425             strcpy(ret_value,dummy);
426         }
427     }
```

```
428
429 bool Function_call::dep_check() {   //check if there exist dependencies
430     if(Dependencies->is_empty()==true)
431         return false;
432     else
433         return true;
434 }
435
436 void Function_call :: constructor_continued(char str[STRLEN],double cont) { //constructor fragmented function calls
437     char dummy[STRLEN];
438     int counter,n;
439     n=counter=0;
440     continued=cont;
441
442     dummy[n++]=str[counter++];      //remove all trash from string
443     dummy[n]='\0';
444     while(strstr(dummy,"= ")==0) {
445         dummy[n++]=str[counter++];
446         dummy[n]='\0';
447     }
448     n=0;
449     while(dummy[strlen(dummy)-1]!='\n') { //read return value
450         dummy[n++]=str[counter++];
451         dummy[n]='\0';
452     }
453     dummy[--n]='\0';
454     ret_value = new char[n];
455     strcpy(ret_value,dummy);    //copy return value
456  }
457
458 bool Function_call :: sim_check(char str[STRLEN]) { //checks whether the name is same and return value is not,
459     if(strstr(str,name)!=0 && strcmp(ret_value,"NOT_SET")==0) //set, I.E. a fragmented function call
460         return true; //fragmented identified
461     else
462         return false; //not fragmented or incorrect name
463 }
464
465 bool Function_call::argument_comparison(char* returnvalue) {
466     Parameter* par; //This function checks whether the return value
467     bool ret=false; //equals the parameter value (dependency)
468
469     for(int i=1;i<=nr;i++) {
470         par= (Parameter*) Parameters->remove(i);
471         if(par->comparison(returnvalue))
472             ret=true;
473         Parameters->add(par);
474     }
475     return ret; //if true, return true, if false return false.
476 }
477
478 char* Function_call::ret_retvalue() {
479     return ret_value;   //returns the return value of function call
480 }
481
```

```
482 void Function_call::add_dependency(int i,char* dep_name,int d_nr, int t, double dep_rtime, char *edge) {    //adds dependency in list
483     Dependency* new_dependency;
484     Dependency* check;
485     char *e;
486     char *str;
487     bool exist=false;
488 
489     if(edge==NULL)
490         e=ret_value;
491     else
492         e=edge;
493 
494     if(Dependencies->in_list(i)) {
495         check = (Dependency*) Dependencies->remove(i);
496         str=check->retval();
497         if(!strcmp(str,e)) {
498             exist=true;
499         }
500         Dependencies->add(check);
501     }
502 
503     if(exist==false) {
504         replace(e,e+strlen(e),'"',' ');
505         new_dependency = new Dependency(i,dep_name,d_nr,t,dep_rtime,e);
506         Dependencies->add(new_dependency);
507     }
508 }
509 
510 
511 void Function_call::write_name(ofstream* out) {
512     int seconds;    //write file name and timestamp (node-name) to .dot file
513     long microseconds;
514     seconds = int(rtime);
515     microseconds = ((rtime-seconds)*1000000);
516 
517     *out << name << "_" << seconds <<"_" <<microseconds << ";\n";
518 }
519 
520 void Function_call::writesubdue_name(ofstream* out) {
521     *out << name << "\n"; //write function call to subdue
522 }
523 
524 void Function_call::write_dep(ofstream* out) {
525     int seconds; //write dependencies (edges) to .dot file
526     long microseconds;
527     seconds = int(rtime);
528     microseconds = ((rtime-seconds)*1000000);
529 
530     Dependency* dep;
531     int tot=Dependencies->no_of_elements();
532 
533     for(int i=1;i<=tot;i++){
534         dep = (Dependency*) Dependencies->remove_no(i);
```

```
        *out << name << "_" << seconds << "_" << microseconds << " -> ";
        dep->write(out);
        Dependencies->add(dep);
    }
}

void Function_call::writesubdue_dep(ofstream* out) {
    Dependency* dep; //write vertices/edges to subdue file
    int tot=Dependencies->no_of_elements();

    for(int i=1;i<=tot;i++){
        dep = (Dependency*) Dependencies->remove_no(i);
        *out << "d " << number << " ";
        dep->writesubdue(out);
        Dependencies->add(dep);
    }
}

Parameter :: Parameter(int nr, char str[STRLEN]) : Num_element(nr) {    //Parameter constructor
    argnr=nr;
    arg_name = new char[strlen(str)+1];
    strcpy(arg_name,str);
}

bool Parameter :: comparison(char* check) { //Check whether parameter name is the same
    if(strcmp(check,arg_name)==0)
        return true;
    else
        return false;
}

Dependency::Dependency(int i, char* dep_name,int nr, int t, double dep_rtime, char* dep_retval) : Num_element(i) {  //constructor
    name = new char[strlen(dep_name)+1];
    type=t;
    d_nr=nr;
    strcpy(name,dep_name);
    ret_value = new char[strlen(dep_retval)+1];
    strcpy(ret_value,dep_retval);
    rtime=dep_rtime;
}

void Dependency::write(ofstream* out) { //Write dependency (edge) to .dot file
    int seconds;
    long microseconds;
    seconds = int(rtime);
    microseconds = ((rtime-seconds)*1000000);

    *out << name << "_" << seconds << "_" <<microseconds << " [label=\"" << ret_value << "\"];\n";
}

void Dependency::writesubdue(ofstream* out) {   //Write dependency (edge) to .dot file
    *out << d_nr  << " " << type << "\n";
}

```

```
int Function_call :: retparnr() { //return parameter nr
    return Parameters->no_of_elements();
}

char* Function_call :: retparstring(int i) { //return parameter string
    char *str;
    Parameter* par;
    par = (Parameter*) Parameters->remove_no(i);
    str = par->retval();
    Parameters->add(par);
    return str;
}

char* Parameter :: retval() { //return parameter name
    return arg_name;
}

char* Dependency :: retval() { //return return value
    return ret_value;
}

bool handlechecker(char* str) { //check whether string is a memory value
    if(strlen(str)>=8 && strstr(str,"0x")!=0) {
        return true;
    }
    else
        return false;
}
```

B Virtualization Scripts

The virtualization scripts were used for automating the dataset environment. By use of these scripts one can execute, parse and log the behavior for an arbitrary number of malware and software.

```
::SCRIPT A:
FORFILES /p D:\malware /c "cmd /c C:\automize_lin.bat @FILE


::SCRIPT B:
set baseline="E:\virtual_machines\rem5\rem5.vmx"
set snapshot=clean1
set dest=/home/rem/
set mal=D:\malware

:: Remove quotes
SET _string=%1
SET _string=###%_string%###
SET _string=%_string:"###=%
SET _string=%_string:###"=%
SET _string=%_string:###=%


mkdir %mal%\%_string%_dir
copy %mal%\%_string% %mal%\%_string%_dir\%_string%

::Trace system call
vmrun -T ws revertToSnapshot %baseline% %snapshot%
vmrun -T ws start %baseline%
vmrun -T ws -gu rem -gp reverse copyFileFromHostToGuest %baseline% "%mal%\%_string%_dir\%_string%" "%dest%malware"
vmrun -T ws -gu rem -gp reverse runProgramInGuest %baseline% -noWait -activeWindow /usr/bin/strace -tt -o %dest%logfile.txt %dest%./ma
TIMEOUT /T 10
vmrun -T ws -gu rem -gp reverse runProgramInGuest %baseline% -noWait -activeWindow /usr/bin/xterm "killall -9 strace"
vmrun -T ws -gu rem -gp reverse copyFileFromGuestToHost %baseline% "%dest%logfile.txt" "%mal%\%_string%_dir\%_string%_system_call.txt"
vmrun -T ws stop %baseline%


::Trace library call
vmrun -T ws revertToSnapshot %baseline% %snapshot%
vmrun -T ws start %baseline%
vmrun -T ws -gu rem -gp reverse copyFileFromHostToGuest %baseline% "%mal%\%_string%_dir\%_string%" "%dest%malware"
vmrun -T ws -gu rem -gp reverse runProgramInGuest %baseline% -noWait -activeWindow /usr/bin/ltrace -tt -o %dest%logfile.txt %dest%./ma
TIMEOUT /T 10
vmrun -T ws -gu rem -gp reverse runProgramInGuest %baseline% -noWait -activeWindow /usr/bin/xterm "killall -9 ltrace"
vmrun -T ws -gu rem -gp reverse copyFileFromGuestToHost %baseline% "%dest%logfile.txt" "%mal%\%_string%_dir\%_string%_library_call.txt
vmrun -T ws stop %baseline%


::Trace hybrid call
vmrun -T ws revertToSnapshot %baseline% %snapshot%
vmrun -T ws start %baseline%
vmrun -T ws -gu rem -gp reverse copyFileFromHostToGuest %baseline% "%mal%\%_string%_dir\%_string%" "%dest%malware"
vmrun -T ws -gu rem -gp reverse runProgramInGuest %baseline% -noWait -activeWindow /usr/bin/ltrace -tt -S -o %dest%logfile.txt %dest%.
TIMEOUT /T 10
vmrun -T ws -gu rem -gp reverse runProgramInGuest %baseline% -noWait -activeWindow /usr/bin/xterm "killall -9 ltrace"
vmrun -T ws -gu rem -gp reverse copyFileFromGuestToHost %baseline% "%dest%logfile.txt" "%mal%\%_string%_dir\%_string%_function_call.tx
vmrun -T ws stop %baseline%
```

C Selected Malware and Software

This appendix contain the selected malware and software for the dataset. It shows the name of all samples, as well as the size of the trace files for the different layers.

Selected Binaries

Name	Func-call size (Byte)	Lib-call size (Byte)	Sys-call size (Byte)
Backdoor.Linux.Blackhole_dir	3	1	3
Backdoor.Linux.BO.c_dir	8	5	4
Backdoor.Linux.BO.d_dir	8	5	4
Backdoor.Linux.Caem.c_dir	4	1	3
Backdoor.Linux.CGI.a_dir	3	1	3
Backdoor.Linux.CGI.b_dir	3	1	3
Backdoor.Linux.Cyrax.b_dir	3	1	3
Backdoor.Linux.DobDrag_dir	3	1	3
Backdoor.Linux.Eko_dir	3	2	3
Backdoor.Linux.Excedoor_dir	3	1	3
Backdoor.Linux.Gummo_dir	3	1	3
Backdoor.Linux.Keitan.a_dir	2	1	2
Backdoor.Linux.Koka.a_dir	6	4	2
Backdoor.Linux.Muench_dir	3	1	3
Backdoor.Linux.Ovason_dir	4	1	3
Backdoor.Linux.Phobi.b_dir	3	1	3
Backdoor.Linux.Phobi.l_dir	10	3	8
Backdoor.Linux.Promptte.a_dir	3	1	3
Backdoor.Linux.Rpctime_dir	3	1	3
Backdoor.Linux.Rst.a_dir	4	1	3
Backdoor.Linux.Sckit.k_dir	3	1	3
Backdoor.Linux.SitC.a_dir	31	6	28
Backdoor.Linux.Smack_dir	3	1	3
Backdoor.Linux.Small.bm_dir	4	2	3
Backdoor.Linux.Small.bq_dir	3	1	3
Backdoor.Linux.Small.bw_dir	4	1	3
Backdoor.Linux.Small.bx_dir	4	1	3
Backdoor.Linux.Small.j_dir	4	1	3
Backdoor.Linux.SSh.c_dir	32	6	28
Backdoor.Linux.Subsevux.a_dir	3	1	3
Backdoor.Linux.UDP.a_dir	3	1	3
DoS.Linux.Arang_dir	3	1	3
DoS.Linux.Chass_dir	3	1	2
DoS.Linux.Icmp.a_dir	3	1	3
DoS.Linux.Icmp.b_dir	3	1	3
DoS.Linux.Icmp.c_dir	3	1	3
DoS.Linux.Icmp.d_dir	3	1	3
DoS.Linux.Igmp.a_dir	3	1	3
DoS.Linux.IISuxor_dir	3	1	3
DoS.Linux.Kod.a_dir	3	1	3
DoS.Linux.Nocwage.a_dir	3	1	3
DoS.Linux.Octopus.a_dir	3	1	3
DoS.Linux.Octopus_dir	3	1	3
DoS.Linux.Overdrop.a_dir	3	1	3
DoS.Linux.Scut.a_dir	3	1	3
DoS.Linux.Slice.b_dir	3	1	3
DoS.Linux.Small.b_dir	3	1	3
DoS.Linux.SSPing.10_dir	3	1	3
DoS.Linux.Stream.b_dir	3	1	3
DoS.Linux.Targ.a_dir	3	1	3
DoS.Linux.Wgcrash.a_dir	3	1	3
Exploit.Linux.Apache.134_dir	3	1	3

Exploit.Linux.Apache.1327_dir	3	1	3
Exploit.Linux.Bind.c_dir	3	1	3
Exploit.Linux.Bnc.a_dir	3	1	3
Exploit.Linux.Bonk.a_dir	2	1	2
Exploit.Linux.Bonk.b_dir	2	1	2
Exploit.Linux.Brk.h_dir	3	1	3
Exploit.Linux.CronDum.b_dir	4	1	4
Exploit.Linux.DCom.e_dir	4	2	3
Exploit.Linux.Espacker_dir	133	131	3
Exploit.Linux.Freeze.a_dir	3	1	3
Exploit.Linux.IIS-Attacker_dir	3	1	3
Exploit.Linux.Ircd.a_dir	3	1	3
Exploit.Linux.Ircd.b_dir	3	1	3
Exploit.Linux.KArtsd_dir	3	1	3
Exploit.Linux.Local.af_dir	3	1	3
Exploit.Linux.Local.f_dir	10	5	5
Exploit.Linux.Local.g_dir	16	2	17
Exploit.Linux.Local.h_dir	4	1	4
Exploit.Linux.Local.i_dir	3	1	3
Exploit.Linux.Local.w_dir	3	1	3
Exploit.Linux.Login_dir	2	1	2
Exploit.Linux.Lpd.c_dir	3	1	3
Exploit.Linux.Mirc.a_dir	3	1	3
Exploit.Linux.Mirc.b_dir	3	1	3
Exploit.Linux.Mms.a_dir	2	1	2
Exploit.Linux.Moogrey_dir	3	1	3
Exploit.Linux.Nuker.Igmp.a_dir	3	1	3
Exploit.Linux.Nuker.Small.b_dir	3	1	3
Exploit.Linux.Nuker.Small.c_dir	3	1	3
Exploit.Linux.Nuker.Small.d_dir	2	1	2
Exploit.Linux.Nuker.Win.a_dir	3	1	3
Exploit.Linux.Nuker.Win.b_dir	3	1	3
Exploit.Linux.Pirch.a_dir	3	1	3
Exploit.Linux.ProcSuid.e_dir	94	1	3
Exploit.Linux.Proftpd.d_dir	3	1	3
Exploit.Linux.Proftpd.e_dir	3	1	3
Exploit.Linux.Race.h_dir	6	4	3
Exploit.Linux.Race.i_dir	6	4	3
Exploit.Linux.Rpc.b_dir	7	2	4
Exploit.Linux.Rpc.c_dir	2	1	2
Exploit.Linux.Rpc.d_dir	3	1	2
Exploit.Linux.Rpc.e_dir	3	1	3
Exploit.Linux.Small.ae_dir	3	1	3
Exploit.Linux.Small.af_dir	4	1	3
Exploit.Linux.Small.as_dir	3	1	3
Exploit.Linux.Small.at_dir	4	1	3
Exploit.Linux.Small.au_dir	11	6	3
Exploit.Linux.Small.k_dir	4	1	3
Exploit.Linux.Small.l_dir	3	1	3
Exploit.Linux.Small.m_dir	3	1	3
Exploit.Linux.Small.r_dir	16	2	17
Exploit.Linux.Small.s_dir	3	1	3
Exploit.Linux.Small.z_dir	4	1	3

Exploit.Linux.Snuq_dir	4	2	3
Exploit.Linux.Sorso.a_dir	2	1	2
Exploit.Linux.Teso.a_dir	3	1	3
Exploit.Linux.Trixack.a_dir	3	2	3
Exploit.Linux.Vma.a_dir	5	2	4
Exploit.Linux.WuFtpd.a_dir	3	1	3
Exploit.Linux.WuFtpd.c_dir	6	1	6
Exploit.Linux.Xpl.a_dir	16	2	17
Flooder.Linux.Alcohol.a_dir	3	1	3
Flooder.Linux.Alcohol.b_dir	3	1	3
Flooder.Linux.Bliz.a_dir	3	1	3
Flooder.Linux.Bloop.a_dir	3	1	3
Flooder.Linux.Echo.a_dir	3	2	3
Flooder.Linux.Echo.b_dir	3	2	3
Flooder.Linux.Fusys.a_dir	3	1	3
Flooder.Linux.Gewse.a_dir	3	1	3
Flooder.Linux.Nestea.a_dir	3	1	3
Flooder.Linux.Pepsy.b_dir	3	2	3
Flooder.Linux.Raped_dir	3	1	3
Flooder.Linux.Rycoll.a_dir	3	1	2
Flooder.Linux.Silly.a_dir	2	1	2
Flooder.Linux.Silly.b_dir	3	1	3
Flooder.Linux.Slice.a_dir	8	3	5
Flooder.Linux.Slice.c_dir	3	1	3
Flooder.Linux.Slice_dir	8	3	5
Flooder.Linux.Small.c_dir	2	1	2
Flooder.Linux.Small.d_dir	5	2	3
Flooder.Linux.Small.e_dir	3	1	3
Flooder.Linux.Small.h_dir	3	1	3
Flooder.Linux.Small.i_dir	3	1	2
Flooder.Linux.Small.j_dir	3	1	3
Flooder.Linux.Small.p_dir	3	1	3
Flooder.Linux.Small.r_dir	2	1	2
Flooder.Linux.Smurf.a_dir	3	1	2
Flooder.Linux.Smurf.b_dir	3	1	2
Flooder.Linux.Stream.a_dir	3	1	3
Flooder.Linux.Synk.a_dir	3	1	3
Flooder.Linux.Synk.b_dir	3	1	3
HackTool.Linux.CleanLog.b_dir	3	1	3
HackTool.Linux.CleanLog.m_dir	3	2	3
HackTool.Linux.Dnstroyer.a_dir	3	1	3
HackTool.Linux.Masan.a_dir	3	1	3
HackTool.Linux.Masan.b_dir	3	1	3
HackTool.Linux.ProcHider.a_dir	2	1	2
HackTool.Linux.Quacker.a_dir	3	1	3
HackTool.Linux.Small.a_dir	6	2	5
HackTool.Linux.Small.b_dir	3	1	3
HackTool.Linux.Sniffer.Sysniff.d_dir	3	1	3
HackTool.Linux.Sniffer.Sysniff_dir	3	1	3
HackTool.Linux.Vulner_dir	3	1	3
Rootkit.Linux.Agent.c_dir	2	1	2
Rootkit.Linux.Agent.d_dir	3	1	3
Rootkit.Linux.Agent.e_dir	3	1	3

Rootkit.Linux.Agent.n_dir	3	1	3
Rootkit.Linux.Agent.sm_dir	2	1	2
Rootkit.Linux.Agent.t_dir	16	13	3
Rootkit.Linux.Agent.v_dir	3	1	3
Rootkit.Linux.Matrics.a_dir	39	11	30
Rootkit.Linux.Matrics.sk_dir	3	1	3
Virus.Linux.Brundle.b_dir	4	1	4
Virus.Linux.Clifax_dir	6	2	4
Virus.Linux.Grip.a_dir	5	1	6
Virus.Linux.Grip.b_dir	5	1	6
Virus.Linux.Grip.c_dir	5	1	6
Virus.Linux.Grip.d_dir	5	1	6
Virus.Linux.Grip.f_dir	5	1	6
Virus.Linux.Little.a_dir	5	1	6
Virus.Linux.Little.b_dir	5	1	6
Virus.Linux.Manpages_dir	3	1	3
Virus.Linux.Nel.a_dir	2	1	2
Virus.Linux.Nuxbee.1403_dir	10	6	5
Virus.Linux.Nuxbee.1411_dir	10	6	5
Virus.Linux.Osf.8759_dir	4	1	3
Virus.Linux.Radix_dir	5	1	4
Virus.Linux.Silvio.a_dir	3	1	3
Virus.Linux.Silvio.b_dir	14	1	5
Virus.Linux.Snoopy.a_dir	35	15	15
Virus.Linux.Snoopy.b_dir	18	8	8
Virus.Linux.Snoopy.c_dir	35	15	15
Virus.Linux.Svat.a_dir	6	2	4
Virus.Linux.Svat.b_dir	4	1	3
Virus.Linux.Svat.c_dir	5	2	4
Virus.Linux.Thebe.b_dir	3	1	3
Virus.Linux.Thou.b_dir	47	40	9
Virus.Linux.Winter.340_dir	11	3	17
bunzip2	4	2	3
bzip2	5	3	3
bzip2recover	3	1	3
cat	8	1	8
chgrp	10	2	10
chmod	10	2	10
chown	10	2	10
chvt	9	1	10
cp	13	2	13
cpio	9	1	10
date	12	3	11
Dbus-cleanup-sockets	4	2	3
Dbus-uuidgen	5	1	6
dd	12	2	13
df	32	20	14
dir	25	12	14
dnsdomainname	6	1	6
dumpkeys	13	2	11
echo	8	1	8
egrep	10	1	10
false	8	1	8

fgconsole	13	2	11
fgrep	10	1	10
fortune	58	35	18
fusermount	3	1	3
getty	9	1	10
Gnome-gnuchess	6	2	5
grep	10	1	10
gzip	6	2	4
hostname	3	1	3
ip	5	1	4
kbd_mode	13	2	11
kill	4	1	4
ln	10	2	10
loadkeys	6	2	4
login	84	67	18
ls	22	8	14
lsmod	103	86	10
mkdir	11	2	12
mknod	11	2	12
mktemp	5	2	3
more	14	3	12
mount	96	84	13
mountpoint	2	1	2
mt	9	1	10
Mt-gnu	9	1	10
mv	13	2	14
nc	14	3	13
open	13	2	12
openvt	13	2	12
ping	4	1	4
ping6	4	1	4
ps	99	25	61
pwd	8	1	8
readlink	10	2	10
rm	10	2	10
runlevel	9	1	11
Run-parts	3	1	3
sed	13	4	10
setfont	13	2	11
sh	17	11	4
sleep	10	2	10
stty	17	8	10
sulogin	5	1	5
swapon	9	1	10
sync	8	1	8
tailf	9	1	10
tempfile	3	1	3
touch	11	2	12
true	8	1	8
udevadm	4	1	5
umount	75	66	10
uname	9	1	8
vdir	57	28	26

wpa_supplicat	14	7	9

D Computational Complexity

This appendix contain data regarding the computational complexity of the trace parser. The appendix lists size of trace files and the execution time required to parse and build the dependency graphs. The goal of this appendix is to provide an indication of the computational complexity of the method, as the runtime increase rapidly with the size of the graph.

Computational Complexity

File size	Calls	Runtime in seconds
320B	4	0,046
501B	6	0,031
1KB	14	0,031
2KB	24	0,046
4KB	52	0,047
8KB	120	0,187
16KB	163	0,639
31,3KB	407	2,059
65,9KB	1092	33,243
124KB	1619	48,066
258KB	4227	898,998
408KB	5721	3205,14