NTNU
Norwegian University of
Science and Technology

Multinomial malware classification based on call graphs

Morten Oscar Østbye

Master in Information Security
Submission date: May 2017
Supervisor: Geir Olav Dyrkolbotn, IIK
Co-supervisor: Katrin Franke, IIK

Norwegian University of Science and Technology
Department of Information Security and Communication

Abstract

Ever since the computer was invented, people have found ways to evolve interaction or simplify tasks with computational resources, this for both good and bad. For the known lifespan of the digital age, malicious software (malware) has been a constant threat to computer systems. Malware has been the cause of enormous damage related to both governmental and private sectors, but also for individuals. Malware has evolved to target different systems and environments and therefore there exists a vastly amount of different samples which differ in both attack methods and functionality. Furthermore, malware has been developed by attackers to exploit unknown vulnerabilities, evade detection techniques and include multiple functionalities, expanding the pool of malware even more. Because of this security expert has to keep up with the development of countermeasures to detect and alarm for this expanding threat.

This thesis addresses one such approach where different malware families are executed, and the traces left from this is analyzed to classify what kind of malware family a sample is. More specifically this thesis utilizes expert knowledge to derive expert graphs describing a malware family, and the graph is then used to match unknown samples to search for likeness.

Sammendrag

Helt siden datamaskinen ble oppfunnet har brukere funnet nye metoder for å utvikle hvordan maskiner kan benyttes til å forenkle oppgaver ved hjelp av ressursene tilgjengelig. Dette har blir gjort med både gode og onde hensikter. Helt siden sarten av den digitale alderen har skadelig programvare (skadevare) blitt utviklet og vært en konstant trussel mot data systemer. Tidligere hendelser har vist at skadevare har potensiale til utgjøre store skader både i offentlig og privat sektor, men også for enkeltsående privat personer. Skadevare har ettersom tiden har gått utviklet seg med tanke på hvile systemer de skal treffe og hvilke funksjonalitet skadevaren innehar. I tillegg til dette har angripere utviklet skadevare til å implmentere forskellige utnyttelser av sårbarheter og unngå deteksjon, dette fører til et enormt antall forskellige typer og variasjoner av skadevare. Også de som forsøker å detektere skadevare og håndtere denne typen programmvare må derfor følge denne utviklingen og forsøke å være føre var på hva som kommer.

Denne oppgaven presenterer en metode som kan benyttes for å skille forskjellige skadevare familier fra sporene som etterlates når de kjøres på en datamaskin. Disse sporene brukes til å analysere skadevaren for å kartlegge hvilken familie den ligner mest på. Mer spesifikt bruker denne oppgaven ekspert kunnskap til å beskrive et utvalg skadevare familer i form av en ekspert graf, denne grafen brukes til å sammenligne ukjent skadevare for å se hvor mye en ukjent skadevare ligner på denne familien.

Acknowledgment

I would like to thank my supervisor, Ass. Prof. Geir Olav Dyrkolbotn and co-supervisor Prof. Katrin Franke, for providing guidance and assistance throughout the project. I especially appreciate how my supervisors has facilitated guidance for me as a distance student at NTNU.

Secondly, I would like to thanks colleagues for rewarding discussions about the master thesis and my employer for support throughout the master period.

Finally, I appreciate all motivation and support from friends and family for throughout the study.

Contents

List of Figures

List of Tables

Glossary

Assembly Code Machine code converted to Assembly language

Binary classification The challenge of classifying instances into one of two classes.

Breakpoint An intentional stopping or pausing place in a program

C++ A programming language

Call Graph A representation of the function call for a program represented in a graph

Deobfuscator A tool with the purpose of counteract obfuscation

Disassembly The conversion of machine code into Assembly code

Graph Matching Comparing graphs to find alikeness

IDA Pro An application for dissembling and debugging of compiled files

MAC address A unique identifier assigned to the network interface for computers

Malware Malicious software

Multinomial classification The challenge of classifying instances into one of the more than two classes.

Non-Sequential Disregard function call order. Simply counts occurrences [9]

Packer A technique to compress or encrypt software

Python A programming language

Sequential Account for the sequence of order

Abbreviations

API Application Programming Interface

ASCII American Standard Code for Information Interchange

COFF Common File Format

DIE Dynamic IDA Enrichment

DLL Dynamic Link Library

DOS Disk Operating System

ELF Executable Linkable Format

GED Graph Edit Distance

I/O Inn / Out

LCS Longest Common Subgraph

MCS Maximum Common Subgraph

NSA National Security Agency

PE Portable Executable

RAT Remote Access Tool

SSH Secure Shell

VM Virtual Machine

1 Introduction

This chapter is an introduction to the thesis, which includes the thesis subject, challenges, keywords, problem description, justification and motivation. The purpose of the chapter is to give an introduction to the reader and point out the thesis contributions and layout.

1.1 Topic covered by the project

For the known lifespan of the digital age, malware has been a threat to all devices, especially those connected to the Internet [4]. And as most of the people today rely on the usage of PC's or other devices to cope with the technological development, the importance of secure platforms are more important than ever. Also, when adding the connectivity and boundless opportunities the Internet provides, everything is set for a malware playground in today's cyber landscape. The information security business is aware of the situation and is always developing malware detection methods, challenging the malware writers to develop their malware to avoid detection. This cat and mouse game is challenging both sides to enhance their creativity and knowledge to write and detect malware [10]. This game has resulted in several thousand possibly malicious samples submitted to anti-virus companies for analysis [11]. Dealing with this increasing amount and diversity of samples has shown manual analysis to be too time-consuming and resource demanding. This challenge has promoted the need for an automated analysis process to be able to cope with the vast number of samples on time [12].

This thesis looks at function call graphs from dynamic analysis of malware samples to detect structural dependencies(graph matching) and similarities between samples.

1.2 Keywords

Malware, classification, intrusion detection system, expert knowledge, multinominal, dynamic analysis, behavioral-based detection, machine learning, function call analysis, dynamic API traces, malware families.

1.3 Problem description

Many attempts are found on using call graphs, extracted from dynamic analysis, to distinguish between malicious and benign code [9, 13, 12]. However, not many researchers focus on distinguishing between malware families using call graphs. Call graph matching, a subcategory of graph matching was successfully used by Lars Arne [9], yet results showed bias due to the different versions of compilers, which was the dominant feature. We can use sample attributes to eliminate this problem. To do so, one can perform automated malware execution while collecting function calls. Further, Neo4j [14] is a prominent tool designed to handle large-scale data. The outcome of the project will be a methodology for multinomial classification of malware families using expert derived call graph matching.

1.4 Justification, motivation and benefits

Malicious software is both insidious and dangerous, and as the world gets more connected through the Internet, the ease of spreading malicious code is increasing. Therefore, the need to analyze different suspicious files to detect maliciousness is more important than ever. In fact, there already exist sophisticated self-modifying viruses [15], as well as tools to quickly produce variations of the same malware [16]. For the analysis of the vast amount of samples wandering the networks, this process is almost bound to be automated for some of the phases in the analysis. Moreover, it is no secret that computer systems play a central part in today's society, covering the needs of a personal computer to the need of controlling critical infrastructure such as electricity, water, sewage and health-care systems just to mention some. Further more, it is proven that targeted attacks can influence such control systems tempering the integrity, availability and confidentiality of these systems.

To correctly label or classify malware at an early stage can give a real insight of the potential damage malware can conduct. Call graph matching could be able to provide this insight [13].

1.5 Research questions

Previous research conducted by Sand [9] used call graphs, extracted during dynamic analysis, for binomial classification. In other words, to decide if an unknown sample is either malicious or benign. This method is also known as call graph matching. This thesis however, is concerned with using call graphs, for multinomial classification, i.e., grouping malware samples into predefined families.

The research questions are presented as a main question with a main goal, and with subquestions (SQ). The subquestions are addressing different sections of research and is showing respectively, requirements (SQ1,SQ2) and evaluation (SQ3).

Main research question: How can existing methodology be used for multinomial malware classification, based on expert driven behavioral call graph matching?

1. **Sub question:** How can function calls with related call values be extracted from a malware dynamically at execution?

2. **Sub question:** How can expert graphs be derived to describe different malware families with expert graphs?
3. **Sub question:** What performance can the described methodology achieve related to classification accuracy?

1.6 Planned contributions

The main goal of this thesis is to classify malware samples into families with the use of expert derived calls graph matching. The classification is multinomial which differs from previous work conducted by L.A. Sand [9]. Furthermore, it explains a methodology for deriving expert call graphs to be used for call graph matching. This thesis also explores the use of Neo4j as a tool for large-scale data analysis.

1.7 Thesis outline

An outline with a brief summary is presented in this section. The content of each chapter is described along with the order of chapters. Firstly an introduction to malware, related literature, call graphs and graph matching is presented. Secondly, the choice of methods along with the conducted experiments is presented.

Chapter 2: Malware
This chapter presents relevant literature about malware detection and classification. A taxonomy of malware is provided along with literature about obfuscation, classification and description of malware analysis in an automated and secure environment. Portable executable (PE) and Executable Linkable Format (ELF) files are described in depth since this is the relevant file format for this thesis.

Chapter 3: Related work
Provides an overview of related literature for the thesis. The specific theory about function calls and how they are related to operating system is presented. Followed by various research for malware detection and classification where call graphs are utilized or classification of malware is conducted.

Chapter 4: Call graph
Presents call graph theory and different approaches for representing call graphs. The chapter covers main definitions used to describe call graphs and is an introduction to call graphs by explaining their origin and connection to function calls and operating system functionality.

Chapter 5: Graph matching
Provides an introduction to graph matching approaches and the challenges of graph matching. And provides definitions for exact and inexact graph matching.

Chapter 6: Methodology for multinomial malware classification
This chapter present and discuss the used methodology for the experiments to be conducted. Firstly the chapter presents a the methodology overview before details and origin of the dataset. Then the extraction of dataset call graphs will be described. Dataset statistics will be presented and lastly the chosen experimental classification algorithm is described.

Chapter 7: Experiment
This chapter presents an experiment to test the feasibility of the proposed methodology and to get and indication on its performance. Moreover, it describes how the expert graphs are derived. Finally, the experiment results are presented and evaluated.

Chapter 8: Discussion
Provides a discussion about considerations and implications the thesis have presented through the experiment. As well as a summary of the thesis.

Chapter 9: Conclusion
Sums up and concludes findings in the thesis.

Chapter 10: Further work
Provides a description of topics that could be further analyzed as further work on the same subject.

2 Malware

2.1 Malware detection and classification

Malware is clearly malicious, hence the name, malicious software. To get an understanding of the term malware, this chapter presents the malware domain. A taxonomy of malware is provided along with literature about obfuscation, classification and description of malware analysis in an automated and secure environment. Moreover, Portable Executable (PE) and Executable Linkable Format (ELF) files are described in depth.

2.1.1 Malware taxonomy

Virus

A virus is a malware that do not have any defined or specified functionality, but which is categorized by the method it is applied to the infected system. A virus can become a part of other programs, and thereby infiltrate other files or processes [17, 18]. A virus can propagate, but it needs some interaction by the infected host to conduct this. Malware is categorized as viruses because it has the same demeanor as biological viruses, thereby the name. The term "computer virus" is probably the most used term by non-technical people for all kinds of malware, due to the relation to a known physical phenomenon. Bishop [17], divides a successful infection from a virus into two phases. The first phase the virus is inserted into the system by hooking into a file, process, registry or boot sector. This phase is the initialization stage where the virus is preparing for the second phase. The second phase is when the virus is executing its functionality. This phase may be whatever the virus is created to perform. Through time virus infections have either spread viral or really made an impact on large systems, so a short description of some legacy viruses that vary in functionality and behavior is presented.

Brian virus

Brain [19] is the name of the first well known virus that infected computers. This virus was released in 1986 and infected the boot sector of MS-DOS computers. Brain spread through writing to all floppy disks that were inserted into a computer infected with the virus, which was quite normal back then when this was the standard way of sharing files between computers.

Macro virus

The first seen macro virus was named "Concept" and appeared in 1995, and infected Word documents, which has been an enormous infection vector the last years as well [20]. These viruses were embedded in word documents and spread by phishing or unknowingly users sending them around. Furthermore, when the document is opened a macro was ran in background executing malicious actions.

Logic bomb

Some viruses have one purpose only, to be destructive. One such virus was observed in March 2013, and where classified as a logic bomb [21]. The virus was spread in a South Korean network for banking and broadcasting. This virus was a logic bomb in the matter that at a certain time it would wipe all hard drives it had access too. This virus successfully took out two broadcasting systems and some ATMs [22]

Worm

A worm, like a virus, is named after a capability linked to the nature of the malware. A worm is categorized by its ability to replicate itself to other networks, clients, server or systems, without any interaction [9, 17, 23]. This kind of malware has the capacity to spread fast worldwide if it utilizes the Internet as is communication form. An example of this is the Sobig worm [24], the worm spread worldwide within hours, when such an infection is happening, it is often referred to as a malware eruption. A virus and a worm are very much alike, but the key difference is that a worm does not need interaction to replicate itself to other systems. A worm may run as a stand-alone process and actively search for other hosts to infect [9].

Tojan

Trojans is malicious programs that masquerades as a benign program. The trojan has its name from the Greek mythology story of Troy, where the famous expression "Trojan horse" was created. The description of a trojan is only the method the malware uses to get access to the system it is about to infect [23]. There are a lot of different methods malware writers can disguise malicious code into programs that seem legit, but some of the more popular methods are to offer free tools or programs such as anti-virus, music/video converters and games [25]. Once such program is installed it may have full benign functionality, or it may pretend to fail, but either way, it has installed the desired malicious components.

Backdoor

A backdoor is a mechanism that allows access to a system in an unusual or alternative way. A backdoor is often a small application or functionality that are activated on an infected host to let an attacker have easy access to the infected system later on. This malware is typically used after a successful exploitation of a vulnerability in an application, making an attacker able to remotely execute code. The remotely executed code is often a small code snippet creating a backdoor for the attacker to gain access to the host. This could be as easy to open for

SSH connections for a specific user assigned by the attacker.

Rootkit

A rootkit is a malware that has implemented functionality to cover itself through manipulation of the infected system. Rootkits have malicious functionality and functionality to cover the maliciousness. The name, "root", reflects that such programs often needs high privileges to perform actions to cover its traces and that a root user can audit all information on a system, while normal users can not. Farmer [26] defines three different types of rootkits; application layer rootkits, library level rootkits and kernel layer rootkits, which operates at different levels in a system.

- Application layer rootkits, which sometimes are referred to as command-level rootkits, has functionality to alter information. This is done by suppressing information from applications or the logs normally available. An example of this is, e.g., to change aliases in a command line for Linux to hide specific files.
- Library layer rootkits. This type of rootkit dives one step deeper. The goal for these type of rootkits is to modify the libraries that application calls to manipulate the information that is provided. An example is to alter the library call open() which normally would open a file. By modifying the library file, the attacker can implement code that will not open the file given as an argument, but rather open another file. Resulting in the library call open(file1) not showing the content of "file1".
- Kernel layer rootkits The main difference for this type of rootkits compared to application and library layer rootkits is that it does not operate in user-land [27], but in kernel-land [28]. A compromised kernel can not be accessed by a user-land process because of memory protection mechanisms and therefore may not be detected from user-land.

Spyware

Spyware is a type of malware with the main purpose of gathering information about the infected computer and the user of the computer [23]. This kind of malware is predefined to collect information such as web browsing history and sends this information back to the attacker who can utilize this information. Spyware is often an effective approach to gathering user accounts and credentials [29].

Adware

Adware is software designed to present advertisement at the infected system either by presenting pop-ups or interacting with browsers to add new startup pages or search engines for the system. A well known form for adware is toolbars, which is a plugin to web browsers where the user gets additional toolbar options for searching [30].

Remote access tool (RAT)

This category of malware is mentioned because it is very often used as a collective name for different malware which offers access to an infected host through a network. There are several remote administrations tools which can be used to cover the core functionality similar to malicious RATs such as AnyDesktop or WinRDP. However, its been a popular term related to malware that offers some remote functionality, through web, console or terminal.

Ransomware

Recent trends in malware show that ransomware has been the most increasing type of infection, this is probably because of the willingness to pay ransom money by the infected users. Ransomware is a type of malware that encrypts files or parts of a system and informs the user what to do to decrypt the encrypted data. It can be compared to a digital kidnapping of information. The last years there has been an increase in the usage of ransomware. In 2015, one ransomware family alone had a total of 325$ million paid in ransom [31], and in 2017 the "WannaCry" [32] attack has showed the impact of such an attack.

2.1.2 Malware obfuscation

Malware is developed to perform harmful actions on a target, this could be various of different actions [33]. Performing these activities will leave evident trails of what has happened and how, this is the core of what malware detection is looking after. For a malware to successfully conduct its operations, it is vital to avoid detection and pass through the security control unnoticed. Eventually, the malware must hide itself to be able to perform its purpose, this can be achieved by obfuscation. As Schiffman [34] quotes ”Obfuscation of malware serves the one ultimate purpose: Survival.”

Malware writers have always been aware that for malware to be able to function over time, it must not be detected. This would give the malware time to serve it purpose by operating, evolving or spreading. So, as the anti virus vendors actively are searching for malware, a sample not concealing itself is rapidly a compromised sample [34]. This would not only spoil the specific attack, but also all other attacks using the same malware.

As Cannell [35] says ”Obfuscation (in the context of software) is a technique that makes binary and textual data unreadable and/or hard to understand”. Obfuscation of code or data is not exclusively a technique used by malware, there are several legitimate reasons for software developers to obfuscate. Some examples are software development related to copyright where applications should only be reserved for one user or machine. Then obfuscation would help to avoid the program to be pirated or reverse-engineered. There are several levels of obfuscation implementation, it may be as advanced as to import cryptographic libraries like AES or Triple-DES, or as simple as an XOR function. Regardless of complexity, one of the main reasons to obfuscate malware is to hide hard coded strings or messages needed for the malware to function [35]. These strings often give away a lot of information about the malware's functionality and origin. Some basic techniques to obfuscate malware strings will be presented.

xor

Malware utilizing obfuscation have been around for a long time, and one of the first examples of this is known as the "Cascade" virus, which was written in assembly. Cascade was notable both because of its visual appearance on the infected systems, but also for using xoring as obfuscation [34]. Cascade was built up with a section of code at the start of the malware that encrypted the oncoming sections of the malware, very similar to how most packers function today. Xor is a simple type of obfuscation, it is a "bitwise" operation which flips a bit to either one or zero depending on the incoming bits, but it is very easy to implement compared to other obfuscation methods such as cryptographic libraries. This because it is a logical operator, implemented in most programming languages. This was a good implementation in the era of Cascade because it altered the malware sample md5sum for each type of xor-keyword key used to obfuscate it. By doing this, the malware avoided detection which was based on pattern matching signatures only. Times have changed and today xor malware obfuscation is not assumed to be a very challenging task to detect for the defending side [36], but it is seen a lot out in the wild, and there's no doubt xor obfuscation makes a malware analysis more time consuming and is easy implement.

Base64

Base64, as xor, is not a complex or very sophisticated obfuscation method that is hard to crack when given time, but it is very effective when simple obfuscation is needed. One of the reasons Base64 have become one of the well known and used methods is because it covers the need of converting binary data, such as machine code, to text format [34]. This means that this obfuscation could be used on systems not able to read binary streams, but just alphabetic characters as many systems are developed to do. One of the big drawbacks with base64 is that is has a very specific padding character "=". This makes the obfuscation easy to recognize. Even though base64 is easy to crack and easy to understand, this could also be to the advantage to the malware writers. By understanding and altering the order of the base64 conversion table, they can alter the obfuscation to their own. This will prevent normal standard decoding algorithms and just as the xor, become a time consuming task to crack.

rot13

ROT is short for "Rotate" which is an assembly instruction that uses letter substitution where the number of letters to rotate is specified. ROT "13" then means that the character should be substituted with the character placed 13 characters after the original character in the alphabet. This obfuscation is a version of the

Caesar Cipher and is a simple letter substitution which in practice uses a table to look up which letter to substitute. This, just like xor and Base64 obfuscation is easy to implement and understand, but do not provide a excellent obfuscation of malware. Although ROT is not an advanced obfuscation, it is used and definitively better than no obfuscation [34].

Xor, base64 and ROT13 are as mentioned three basic technique used to obfuscate malware strings and provides to some extent the malware obfuscation. Encrypting is often cited as another method for malware to avoid detection. Even though encryption does the same thing, it distinguishes itself from obfuscation. When encrypting, data are confidentiality held even if intruders know the encryption algorithm, the secret is kept with the key. By obfuscation, an intruder will be able to gain knowledge of the original data as long as one knows the obfuscation algorithm [37]. This is where encryption and obfuscation differs.

Olgiomorphic, Polymorphic And Metamorphic Malware

Morphic code is a name for code that alters code. This is a very broad definition, but it describes the main principle with morphic code. The reason malware writers implemented this type of code is mainly to avoid signature based detection [34]. A malware which alters itself, even by just a byte, alter the hash of the sample significantly. The implementation varies a lot in sophistication, but there are three main types of morphic code; Oligomorphic, Polymorphic and Metamorphic code. All three methods will be presented in this section.

Oligomorphic

Oligomorphic was one of the first countermeasures malware writers used to avoid pattern recognition detection. Oligomorphic is a Greek expression meaning few-shape, in this case referring to which extent a malware alters itself. Oligomorphic malware has an encryptor that is differently implemented for each sample, thereby altering the malware for each time it is compiled. In oligomorphic malware, the malicious code "body" was unchanged, but some parameters to the encryption algorithm or the encryption algorithm in itself were changed. Detection of oligomorphic malware uses different keys or a handful of different encoders is manageable to write pattern based signatures to detect [38]. An example of oligomorphic malware was a virus found in the wild in the 90's named "Whale", this was the first of its kind and choose a random encryption key to obfuscate itself. Other, later versions of oligomorphic malware use dynamically changing encryption making it very hard to write pattern based signatures for detection. However, oligomorphic was only the tip of the iceberg, that would lead on to become polymorphic malware.

polymorphic

Polymorphic functionality is the next level of making malware a pain, the cornerstone in polymorphic is the same as in oligomorphic, and is based on malware changing itself or evolving. The meaning of the word “Poly” is “many” and refers to that polymorphic malware makes many changes to itself while keeping the original functionality. The transformation for each iteration of the malware is drastically bigger than in oligomorphic, and it would be very hard to see any correlation in the encryption [34]. One of the first well known polymorphic malware was written as a project to prove that signatures checking for patterns were not adequate for malware detection. It was called “12” AKA V2PX” and used methods to include useless instructions or registry entries in the code dynamically to alter the malicious code, but not its functionality. Some examples of this is to use an algorithm to insert different “nop” or “clc” instructions to obscure itself. To counteract this evolution of malware obfuscation, the defenders started to run suspicious files in closed environments to monitor the behavior of the sample instead of looking at its patterns. This has been known as the art of behavioral analysis, and promoted the use of sandboxes [39].

Metamorphic

Even though polymorphic malware made analysis more difficult, the malware writers escalated the arms race quickly, by introducing a new malware obfuscation method called metamorphism. “Meta” means meaning about itself, and this is where the malware type stands out. This type of malware introduced the ability to understand its code and meaning, and reprogram itself to functionally operate likewise, but use different approaches to the functionality. This technique was also referred to as “body-polymorphism” According to Walenstein et.al [40] two main principles classifies metamorphic malware: The way they transform and the way they communicate. Transformation is divided into two subparts, one is “Binary transformation” this is where the binary of the executable itself mutates. The other is called “alternate representation transformation” this is when the mutation is based on the understanding of its code. The malware reads pseudo-code of itself and evolves based on the code. Regarding communication, the classification is also divided into two subparts, “open world” and “closed world”. This, as the names intend, refers to if the malware can connect to other systems to download extra features like plugins or not.

Packed malware

A packer is as Axelle describes, “System used to compress and encrypt software [36]". And there are several legitimate and useful ways to use a packer. Packers have been used to compress the size of files, this was originally implemented in the area when disk size and bytes was a concern. This is not as relevant anymore, but some developers use packing to protect or hide copyrighted code, which is highly relevant today. A packer often contains a “Packer stub”, this stub executes the unpacking of the malware as shown in figure 1. This kind of code hiding, of course also triggered malware writers interests. Malware packing has been a popular method to avoid detection because different packers will give the same malware a different hash value. Also, when implementing a packer, a malware writer would just need to alter a small amount of code in the packer to make big changes in the total file, instead of changing significant parts of the malware body. There are several, if not endless ways of packing a malware. Some popular

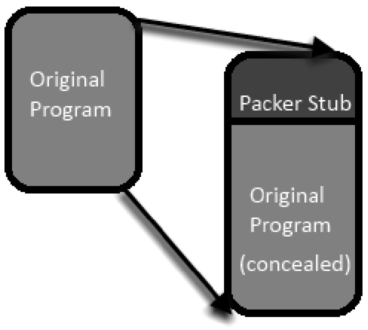


Figure 1: Malware packer stub

packers and methods are UPX, Polypack and Themida. Polypack was an application that would pack a file with different packers and scanned them with anti virus scanners to determine if it would be detected or not. This would for obvious reasons come in handy for malware writers that want to pack their malware and test if anti virus would detect the packed version. As the number of different packers increased, the defending side started gathering signatures to detect them. This signature detection is searching for bytes that are characteristics for the different packers, and a single rule would be needed for each packer. This resulted in long lists of signatures, and today there is available signatures list that covers over 10 000 different packer varieties easily available. The reason this could be done is that a packer stub would be in "plain text" as the machine would have to understand the decryption algorithm. A popular packer detection tool is PEid, and covers many of the widely known and used packers, but as always, a minor change to the packer would avoid the signature based detection. Oberheide [41] performed packed malware testing of 98,801 samples, and found that 40 % them were packed, which gives an insight of the broad usage of packing.

2.1.3 Malware classification

Classification of malware is presented to describe the functionality or behavior of a sample. The description of different malware in section 2.1.1 is the traditional division, but as threat actors are developing their technical and tactical procedures, classification of samples has to adapt over time. An advanced threat actor often delivers malware in several stages to cover capabilities or to customize the attack for the victim. To describe the different stages of an attack chain an alternative division of classification may suit better [42]. Some of the widely used taxonomy is described in this section. It is important to point out that this is after the initial exploit stage in an attack.

- **Downloader**, is a typical first stage malware. This type of malware initiates a connection to download and execute additional malware. Some downloaders have the capability to perform privilege escalation before execution of the downloaded malware. Examples of downloaders can be pdfs, office files, executable files with mentioned functionality implemented.
- **Dopper**, is also a typical first stage malware, which has the goal to execute additional malware. The difference from the downloader is that a dropper has the next stage malware integrated into the dropper. The malware drops a file to memory or disk which is embedded in the dropper. An example of this is to drop a file from the resource added to an executable.
- **Agent/Implant**, is a typical second stage malware which calls home and can receive command and control communication. This stage often serves limited functionality but can be extended with additional functionality. This was a term origination from a NSA leakage [43].
- **Stager**, is an implant variant, which is a term origination from the metasploit framework [44]. This is a payload delivery option where each command from the attacker is loaded with a new stage to be executed by the stager.

Malware families

Malware family affiliation is a concept used to name a sample to allow easy identification of function and purpose. Obtaining an understanding of the relationships between individual pieces of malware provides insight to the analyst. The assignment to families is not class discriminating, different malware and sometimes non-malicious samples, can be assigned to the same family [45]. For a sample to be assigned to a family, it needs to have some relation to the other family members. The relation of a sample to a family can quite subtle [46], this is a non exhaustive list of attributes that can classify a sample to a family [46, 47].

- **Attack specific malware**, is malware used in the same attack chain, or similar attack chains.
- **Actor specific malware**, is malware used by the same actor attributed to a specific family.
- **Functionality specific malware**, such as Banking Trojans family. Where the family affiliation is determined by the functionality.
- **Code specific malware**, is malware where the code of the malware links it to a family. This could be specific code sections in the malware which are

reused. This is often linked to actor specific malware.
- **Campaign specific malware**, is the case of different malware variants is used in the same attack campaign.

For this thesis, the term family is related to functionality specific malware.

2.1.4 Malware Analysis

Malware analysis is the study of malicious software that includes different tasks that can be grouped into several stages or areas of study. A widely used partition is statical and dynamic analysis [4]. The complexity of the steps in the different methods differs, Zeltser [1] has described, a pyramid for the stages an analysis method can be divided into and its complexity. The model consists of the stages, manual coding reversing, interactive behavior analysis, static properties analysis, and fully-automated analysis. These will be described along with the main classification static and dynamic.

- **Static malware analysis** [48] is usually performed by dissecting the malware and searching for different artifacts about the program, and analyzing them part by part. This could be investigating the binary, searching for strings to reveal the functionality, behavioral or intent of the malware. Alternatively, disassembling it and read the machine code associated with the file to understand its functionality. The cornerstone of this approach is that the malware is not executed. This conserves the analyzing environment for the actions of malware, which is a big advantage.
- **Dynamic malware analysis** [48] is conducted by running the malware and look at the actions it performs. This could be done by attaching a debugger and step through the program or execute it when logging is enabled to log the actions. This form of analysis is often faster than static, but it produces an enormous load of information that needs to be analyzed, so log filtering is crucial when conducting a dynamic analysis.

Four stages of malware analysis

1. **Fully automated Analysis** This is regarded as the easiest way of analyzing a malicious file, this is refereed to as the usage of tools or systems available, not creating the environment. The simplicity is a result of what the systems are looking at what the specimen is doing when it is executed on a system. This often gives the specimen a rating of badness and generates a report to serve to an analyst. These reports often include malicious artifacts like network traffic, file altering, mutexes and registry changes. These fully automated tools are often not as comprehensive and give the insight of what a human analyst could give. The upside of the method is that is very quick, which is crucial when many samples need to be analyzed. Not to mention that malware analysis is a time consuming activity and often

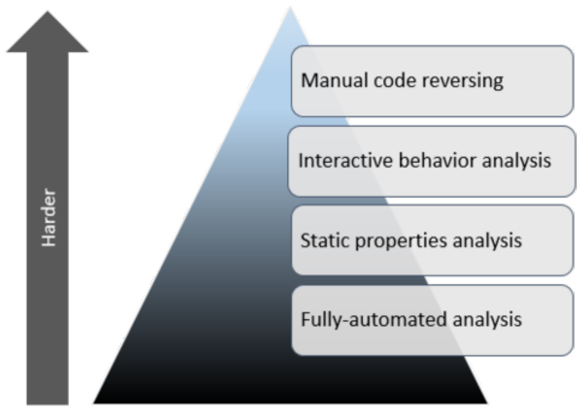


Figure 2: Four stages of malware analysis [1]

expensive since trained analysts are difficult to get a hand off.

2. **Static Properties Analysis** After an automated analysis, a malware can be flagged as suspicious. A closer analysis should then be conducted for the file, searching for static properties is a natural next step. This is also a quite fast and easy understandable analysis and often consists of looking at the properties of the file including header, trailer, compilation-times, file authenticators(hashes), imported/exports functions and ASCII string inside the file. This could reveal much information about the file, but an analyst must have some experience to know what to look for. This stage has a weakness against malware that is obfuscated or packed, but there are static tools that can detect if a file is packed by looking at artifacts as entropy in the sections of the file or using signature packer detection. This is considered as static analysis.

3. **Interactive Behavior Analysis** The next level of a malware analysis could be to conduct a behavioral analysis, this is considered as a more complex and comprehensive method to analyze the sample than the previous methods. This method often takes the sample into some laboratory or protected environment and tests the malware. This is a considered as dynamic analysis. The implementation of this approach, as in dynamic analysis, it to set up a logging environment to log the behavioral and actions of the sample to understand its intended actions. This could, for example, be to perform a network packet capture to look for command and control traffic from the sample. The same challenges with information overload are associated with this approach as mentioned in dynamic analysis.

4. **Manual Code Reversing** Reverse engineering is the art of taking an object and take it apart to understand how it works and how it is built up. This can also be done with malware. When a malware is compiled from readable code to a binary file, the reversion of this process is partly completable. A binary could be disassembled to assembly code. This is the most used way of manual code reversing, even though there are countless malware that is not compiled but needs to be reversed to understand its functionality. This method of malware analysis is seen as very time consuming and difficult to perform. Good knowledge of programming and system configuration is needed to understand how the samples are operating. Eventhough the method is resource consuming this is often the only way to understand how data from malware is encrypted, how algorithms the malware uses is implemented and understanding the capabilities of the malware.

2.1.5 Automated Analysis

Automated analysis of malware, is as mentioned earlier, an easy and fast way to conduct initial analysis of malware, and several commercial tools offer this functionality today [49]. Automated analysis is often related to, or uses a sandbox, and is therefore described in briefly in this paper. As SecTechno [2] describes, automated analysis should be conducted at an initial stage. This process allows the system to analyze a new binary in a newly setup environment, also known as an automated sandbox. The cycle has 7 cyclic steps as shown in Figure 11 and one initialization step. The initial step when applying an automated analysis is to define a baseline, this is a process of listing different artifacts that the system should know as a base, and what to not look for. This could be anything the sandbox want to analyze, but popular artifact is names, registers, files, timestamps, hashes, memory, function calls, network activity, execution of files. This artifact should then be monitored, but the system should also be able to differ the abnormal form the normal, which is the main point of a baseline. The steps for the automated analysis cycle is as presented.

1. **Revert / re-image the target** This step is meant to make the environment ready for the malware analysis. This would be re-imaging of machines, or reverting virtual machines to a clean state. It is important that the state of the clean machine is as close to the baseline as possible, so the analysis do not have to consider normal activity of the operation system as anomalies from the baseline and to mark them as malicious or suspicious.
2. **Transfer the malware** The cornerstone of this step is to get the malware into the analyzing environment. This phase can be conducted in various ways, but the importance of doing this safely should not be underestimated. Transferring the malware in a shared network folder opens an exfiltration vector for the malware. It is important that the analysis environment can control the malware in a safe manner. The transfer could be done

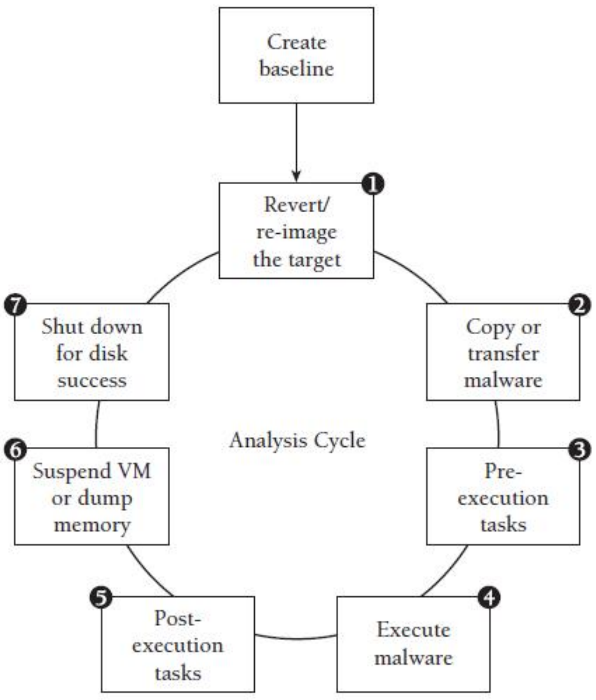


Figure 3: Visualization of the 7 steps of malware atomization [2]

by copying the file, sharing read-only folders or using client executing applications like PsExex.

3. **Pre execution tasks** This step should get the environment ready for the execution of the binary, but it also performs some static analysis of the binary like looking for suspicious strings, function calls, taking snapshots of register or hashes of the binary. This step could also be to check that the environment has the right access to run the malware or that Internet access is provided if necessary.
4. **Execution of the malware** This step must be malware type aware, in the context of what file type the malware is, and adapt an environment to the file specific prerequisites. There may be some prerequisites for the binary to be able to run, such as command line options, or if the binary is a .dll. This challenge should be considered in this step.
5. **Post execution tasks** This step identifies the changes the binary has done to the system. This could include operations such as taking screen shots, register snapshots to compare with before the execution, running tools like process monitor and other system artifact investigating tools, and collecting logs.
6. **Suspend VM or dump memory** To obtain a memory dump of the system,

the memory must be dumped. This could be achieved by suspending the virtual machine and copy the memory file or dumping the memory of a physical machine over the network or to a hard drive.

7. **Shut down for disk success** The last step is to log for differences on the hard drive from the baseline and after the execution. Therefore the disk needs to be shut down to analyze the alteration the binary has done. This is also to release resources for analyzing the next binary that is to be a part of the cycle.

2.1.6 Secure Environment

When using sandboxing in an automated analysis to handle malicious software, securing the environment is a crucial task to prevent the malware to perform unwanted actions to the systems analyzing the software. Thereby, securing the environment is an impotent aspect when dealing with sandboxing, either it is manually operated or automated [39]. Three widely used technical solutions for handling this challenge is the usage of physical hosts, virtual machines or a controller.

- When using a physical host re-imaging the machine after analysis is part of the analysis cycle, this could be very time consuming and resource demanding. The pro of this technique is that it is a very authentic environment for malware that is aware of virtualization.
- The usage of virtual hosts, this is fast and have the opportunity of snapshots to revert the machine to an original state. However, some malware is virtualization-aware, and would not run in a virtual environment.
- Finally the usage of a controller can be implemented, this is a technique where there is a controller machine that handles the environment. This can be an environment of several sandboxes where both virtual and physical machines are involved among other networking devices.

Keeping The Environment Secure

To maintain the environment secure, not to let the malware run uncontrolled, it is important that the software is updated, and to limit the permissions the malware has and to control network cards. It is important to have control of removable media and shared folders malware can use. To control and monitor the environment, a firewall and IDS should be implemented along with suitable network configuration. This could also be used to direct the network traffic to monitor nodes [39]. An example of this is if you are simulating a network service to answer command and control traffic from malware.

Typical Limitations

Typical limitations for sandboxing is the same limitations as in most information security infrastructure which is the cost of the system. One of the problems con-

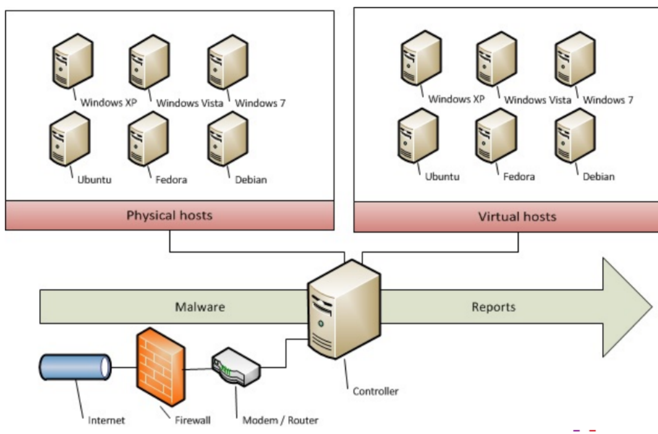


Figure 4: Vizualization of secure environment with controller [3]

nected to sandboxing is operating system licenses, since a sandbox appliance to be able to handle several samples at the same time, they need to be executed in different environments. This means that each environment needs, e.g., a Windows license to run. This is often a significant cost for sandboxes. Moreover, even if there are no host limitations, there is a lot of malware that is acting differently depending on the system configuration such as Java version or flash version. Then to be able to analyze malware the sample must be executed in different environments with different configurations. An approach to reducing the cost of this is to limit the various configurations to the protection environment the sandbox is protecting, but this is not able to analyze all the different variations of malware.

2.1.7 Anti Reversing

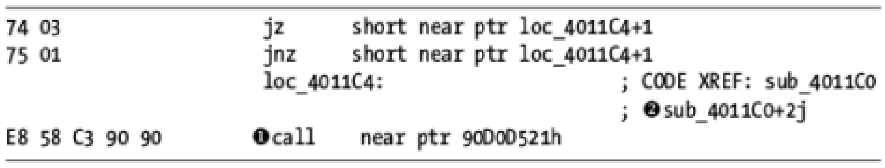
"Reverse engineering is known as the process of extracting knowledge or design information from anything man-made and reproducing it or reproducing anything based on the extracted information" [50]. Reverse engineering is a complex working methodology to conduct, especially when it comes to software reverse engineering. The expertise of a trained reverse engineering is a very valuable part of a malware analysis. In this case, reversing due to malware analysis will be the task of dissecting a piece of malware to understand its purpose and behavior. There are many different approaches to reverse engineering and just as many techniques to avoid or stop them. Some of the most used reverse engineering techniques are disassembly, debugging and virtual machine techniques. This part of the chapter will shortly present the different methods and how to prohibit these types of reverse engineering techniques.

Anti Disassembly

Disassembly is the art of "tearing" a program apart into pieces. This needs to be done because when programmers are writing code, they in most cases write in a language readable by humans such as c++ or python. This is considered as high-level languages, this code needs to be compiled into a language machine can read. This is when compilers transform the readable code to a less human readable format machine code [4]. When a reverse engineering is trying to reverse a program is normal to parse the binary file(compiled code) in a application(e.g., IDA pro) which can present the machine code instructions. This is very often presented in assembly code. The reading of the binary file is where anti disassembly is trying to interrupt, this to make the application presenting the machine code faultily. This can be done in different ways, and some of them will be presented. The cornerstone and the goal of the various implementations are to produce an incorrect or wrong listing of the program represented and is mainly achieved by implementing specially crafted code. This could result in application error. Moreover, it is very often this method is implemented to waste the time of the analyst such that the malware can operate some time before different characteristics that reveal the malware behavior is found. As mentioned, anti disassembly uses techniques to fool the disassemblers, this is done by looking at the assumptions the disassemblers do and how these assumptions restrict its functionality.

Jump conditions with the same target

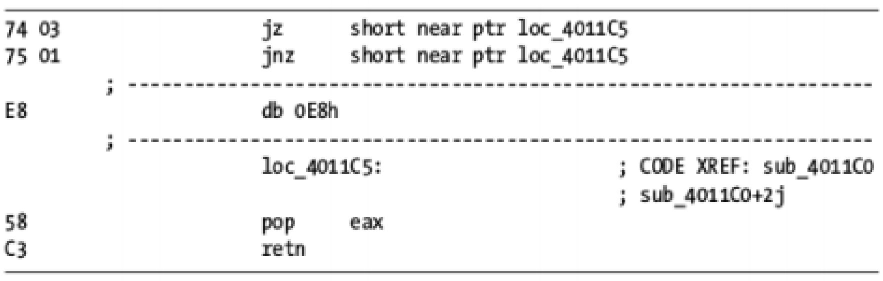
A known used technique used to make malware analysis more difficult is when several jump conditions uses the same jump target. This is often implemented by two conditional jump instructions, e.g., jz and jnz, these two instructions will together be placed right after each other, resulting in a unconditional jump. This means the jump will be carried out regardless of what the value of the tested flag is. Also, even though one of these branches in this setting will not be true, it will be disassembled since the opposite jump condition will lead the flow of the program down that branch [4]. This could be implemented along with an inaccurate jump like shown in the code in Figure 5. In this code, it looks like there



```
74 03              jz      short near ptr loc_4011C4+1
75 01              jnz     short near ptr loc_4011C4+1
                   loc_4011C4:             ; CODE XREF: sub_4011C0
                                           ; ❷sub_4011C0+2j
E8 58 C3 90 90    ❶call    near ptr 90D0D521h
```

Figure 5: Anti Disassembly jump to same location assembly, [4] page 335

is a conditional jump, and that is right, but together it is an unconditional jump. Moreover, when looking at where the jump is referring to "LOC 4011C4+1" this looks like "call" instruction that begins with the with a value 0xE8, but when the disassembler reads this is does not take into account that the first byte is not "just" data and not included when converting the data to functional code. When using an instruction to data conversion function at the first byte, and then letting the disassembler read on the outcome is quite different. Then another succeeding array of instructions is presented. This is shown in 6.



```
74 03              jz      short near ptr loc_4011C5
75 01              jnz     short near ptr loc_4011C5
        ; ---------------------------------------------------------------
E8                 db 0E8h
        ; ---------------------------------------------------------------
                   loc_4011C5:                 ; CODE XREF: sub_4011C0
                                               ; sub_4011C0+2j
58                 pop     eax
C3                 retn
```

Figure 6: Anti Disassembly jump to same location assembly with real code, [4] page 335

Impossible Disassembly

Impossible disassembly is a section of bytes which can not be represented correctly by any disassemblers. The example in Figure 6 shows a pseudo impossible disassembly technique, where there was a "data" byte squeezed in at the start of the next instruction, tricking the disassembler to misinterpret the instructions. The "data" byte in examples as this is called a "rogue byte". The same thought is used in impossible disassembly, but with this technique the rouge byte is not just some data, it is needed to successfully continue the disassembling [4]. In other words, the rouge byte can not be ignored. This can be achieved if a byte is a part of two or more instructions. An example of this is if a jump instruction is jumping inwards into itself, and the half of the instruction is used in the next instruction. As shown in Figure 7 there is a byte string "EB FF C0 48" where the "EB FF" is representing a jump inwards into itself, and then the "FF C0" is representing an "inc eax" instruction before a "48" is representing a "dec eax". This will confuse the dissembler and faulty code will be presented.

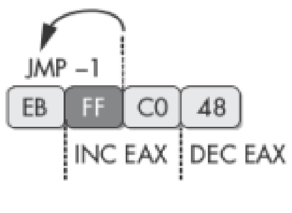
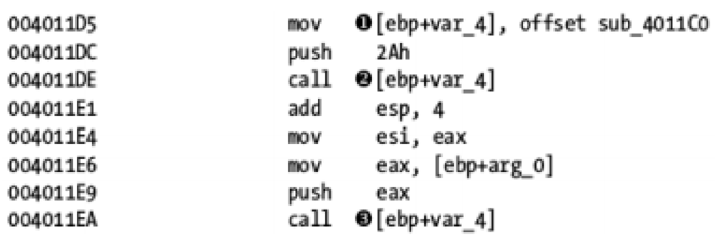


Figure 7: Anti Disassembly inward jump to obscure program flow, [4] page 337

Function Pointer Problem

Malware writers can use the way disassemblers is correlating functions to obscure the flow of a program, one way to do this is to refer to an offset multiple times using a variable to store the offset. By doing this, disassemblers would only be able to see one cross reference, and this would be the initial reference when moving the offset into the variable. If the variable then is called later on in the subroutine the disassembler would not be able to detect the oncoming cross references. This is because of how the disassembler is loading the variables onto the stack. It would assume that the call to a variable is just a call to that variable and not a reference to another offset. This is visualized in Figure 8 where the cross reference is correct at point 1, but is missing when referring to the variable at point 2 and 3.

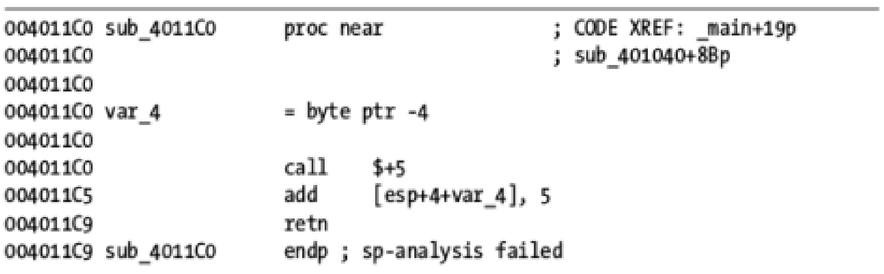


```
004011D5        mov   ❶[ebp+var_4], offset sub_4011C0
004011DC        push    2Ah
004011DE        call  ❷[ebp+var_4]
004011E1        add     esp, 4
004011E4        mov     esi, eax
004011E6        mov     eax, [ebp+arg_0]
004011E9        push    eax
004011EA        call  ❸[ebp+var_4]
```

Figure 8: Anti Disassembly function pointer problem, [4] page 341

Return Pointer Abuse

Another way to interact with the flow of a program is to take advantage of how the return pointer is working. For all purposes, the return pointer is used at the end of a call function when the flow of the program is returning to the ongoing flow before the subroutine was called. The reference of the return pointer is pushed onto the stack when entering a subroutine and will be jumped to when returning(retn). A retn is basically a “pop” of the stack and a “jmp” to that value. An anti disassembly technique would be to confuse the disassembler to make a “retn” call jump to a specific place. In Figure 9, an example of how a “retn” can be used to do this is visualized. The three first instructions in Figure 9 is the reason this technique is possible. The first one “call [$]+5” is used to call the location following itself, this location points to a location in the memory that is being placed on the stack. After this instruction is done, it puts the value “004011C5” on the top of the stack. Following is the instruction “[esp+4+var 4], 5”, this instruction looks like it is referring to the variable “var 4”, but this is not correct. When looking at the declaration of “var 4”, it is clear that “var 4” has the value negative 4. This means the full instruction will come together as “[esp+4+-4], 5” or just “[esp+0], 5”. So this instruction is adding 5 to the to the value at the top of the stack. When then the last instruction is executed, the “retn”, the newly altered value positioned at the top of the stack is popped and jump to by the “retn”. When writing code like this, it is possible to write confusing code that needs to be carefully analyzed to be certain where the return instruction is referring to. This makes the analysis process harder and more time consuming.



```
004011C0 sub_4011C0      proc near             ; CODE XREF: _main+19p
004011C0                                       ; sub_401040+8Bp
004011C0
004011C0 var_4           = byte ptr -4
004011C0
004011C0                 call    $+5
004011C5                 add     [esp+4+var_4], 5
004011C9                 retn
004011C9 sub_4011C0      endp ; sp-analysis failed
```

Figure 9: Anti Disassembly return pointer abuse, [4] page 342 - 343

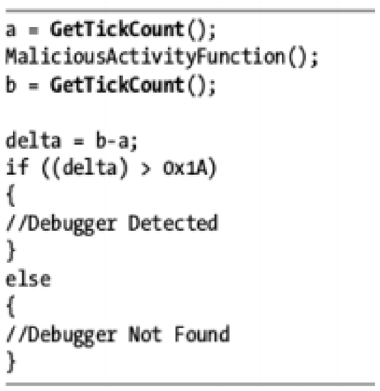
Anti Debugging
For this section, different methods and techniques for implementing anti debugging measures will be presented. Debugging is a process to locate, overcome and eventually fix errors or bugs in software. This is done by running the program with a debugger attached, doing this it is possible to stop the program at predefined locations, or step through the program instruction at instruction while looking at different values in the stack and memory [4]. This is a very valuable approach when dissecting malware, to understand how it works and how to extract special artifacts belonging to that exact malware. Therefore malware writers have a great desire to detect if the malware is attached to a debugger, so different methods to detect this and anti debugging measures have been applied to malware over time. Some of the methods to detect a debugger or debugging behavioral will be presented.

Breakpoints
Breakpoints is a widely used method to debug programs, its main area of usage it to set breakpoints to break the program at certain points where the code needs a closer examination [51]. The person debugging can then look at the values linked to the code and step instruction by instruction from there on. There are three different methods to insert a breakpoint, these will be presented along with how to detect them.
INT 3, is the most common type of breakpoint, this is an instruction in the Intel x86 instruction set and generates a software interrupt to stop or pause the program [51]. The opcode or instruction for an INT 3 is “0xCC” which is quite easy to recognize, for a malware writer. So, if a malware wants to check if there is sat any INT 3 breakpoints, it could search the code for “0xCC” values. By doing this, the program can act differently then what is normally does when it is not

attached a debugger and breakpoints, or it can quit.
Hardware Breakpoint as the name intend, breakpoints are sat by the hardware. The breakpoints are implemented in the Intel processor architecture, and there are specific registers sat to perform these breakpoints. These are named Dr0 - Dr7 and the different numbers shows to different properties linked to the different breakpoints. As these breakpoints depend on hardware, there is a limit to how many of them it is possible to use. This number of possible breakpoints is normally two or three [4]. Detecting these breakpoints is quite easy, and can be done by the use of the Windows function “GetThreadContext”. By retrieving a handle to the current thread it is possible to check which registers is sat, and if a malware writer checks for the register values or Dr0 through Dr8 it can tell if hardware breakpoints are sat or not.
Memory Breakpoints, is the "go to" implementation of breakpoints when a malware analyst is expecting a malware sample that has implemented anti debugging. This because there is often more difficult to detect a memory breakpoint, there is no direct way of doing it other then checking for the same exceptions as the debugger would check for itself. To set a memory breakpoint the debugger is using what is called a “PAGE GUARD”, the guard is sat at the memory location the breakpoint is desired. Moreover, when this memory location is addressed, an exception is raised, and the program breaks. For a debugger to be able to do this, functions for setting the guard and handling it are necessary [51]. If a malware writer wants to check for this, the same functions and approach must be implemented to detect it. However, it can be conducted by marking memory pages as guarded and push a return address onto the stack before jumping to the guarded page. By doing this, if the program is under a debugger it will return to the address pushed to the stack, because that is how the page violation function works. The other outcome is that a genuine guard page violation is occurring and then the page was “clean” initially, and there is no breakpoint present.

Timing Attacks

Timing attacks is a very popular type of debugger detection, this mainly because it is quite easy to implement and that it works effectively. Even though the implementation of a timing attack often is uncomplicated, the detection consists of three parts and can easily be overlooked. This because the different sections can be placed in different parts of the code and can have a low amount of instructions all together [4]. The main thought with a timing attack is that the malware is aware of how long time the malware have been running, and as it is known roughly how long time it would take to conduct the actions it is supposed to, the malware can be conscious about it. This means that the malware can start a counter before it performs a task or subroutine it knowns the time required to conduct, and then check the time again after the task or subroutine is conducted and subtract these numbers to get a result. If a debugger is attached and a malware analyst is stepping through the program the time consumed would be a lot longer. Moreover, by implementing a counter check of the result, the program can be aware if a debugger in attached or not. There are several ways to get the time in an operating system, but some of the most used are the “rdtsc” instruction which gets the number of “ticks”, which the speed a computer system clock is running at, and check this before and after. The function “GetTickCount” can also be used, but the approach will be the same. A visualization of a time attack is shown in 10. Where a "Tick count" is started before the malware executes a function with malicious activity. After the function is finished a new "Tick count" is performed, and subsequently the two "Tick counts" are compared.



```
a = GetTickCount();
MaliciousActivityFunction();
b = GetTickCount();

delta = b-a;
if ((delta) > 0x1A)
{
//Debugger Detected
}
else
{
//Debugger Not Found
}
```

Figure 10: Timing attack example code, [4] page 359

Windows API

When a debugger is attached to a program is leaves some traces, and as a malware writer, it is several Windows races to look for to detect if a debugger is present. The Windows API offers some functionality making this easy, these functions were developed to help the usage of debuggers and most of them is documented [4]. The following functions are some of the API calls that could be used.

IsDebuggerPresent

This is a function to search through the process environment block and checks it for a field named “IsDebugged”, if it is found the functions will return a nonzero value if it is found and zero if it is not. This value can then be tested by the malware to see if it is attached to a debugger or not.

CheckRemoteDebuggerPresent

This is a function checking for the same at “IsDebuggerPresent” the only difference is that this function can be handled other processes handles to check as well, therefore the “remote” part of the function name.

NtQueryInformationProcess
This function retrieves information about a process, and inside this information, it checks for values associated with debugging, like “ProcessDebugPort” and if it finds a match it returns a port number, otherwise it returns zero.

OutputDebugString
This is a method to check if a debugger is present by trying to send a message to the debugger. If the message is sent to a debugger successfully, there is no doubt there is one attached. The function will give an error value if the function fails and it does not manage to send the message to the debugger.

Code Checksums
Checksums are often implemented to check for errors in transmission, to ensure that the packet is received exactly as it were sent. This method can be used to see if someone is altering the program someone has written. By checking the checksum of parts of the original code and compare it at runtime, it it possible to check if there have been added a breakpoint in the form of a instruction(0xCC) or something else. This could then be a test for malware to see if someone is trying to add or remove parts of the code to understand it. This could, for example, be a simple md5sum check of the sections of the code. This would not detect hardware breakpoints since these will not alter the code.

Anti virtual machine techniques
Malware analysis often includes using a virtual machine when analyzing a malware sample, this because virtual machine hardware can isolate the malware and have useful functionality like snapshots. This functionality makes it possible to restore the state of a machine to a previous state, by doing this malware analysts have the possibility to perform the same task several times and focus on different artifacts of the behavioral at the time, making the analysis much easier. So, malware could have a benefit of detecting if it is being executed in a virtual environment. This could imply that the malware is being analyzed, and it could then perform misleading actions or quit if it does not want to be executed in a virtual environment. This section will present different anti virtual machine techniques.

Detecting VMware artifacts
When a machine is running in a virtual environment, a lot of characteristics are left behind, especially if the virtual machine have installed VMware tools [4]. This applies not only to VMware, but also other virtualization software, and the principle would be the same, so for the simplicity, VMware is used as an example. Some of the characteristics the machine will leave are different names of processes, registers and files. This can easily be searched for by the malware, and some examples can be services running on the hosted named “VMware-

Servce.exe” or “WMwareTray.exe”. These are quite obvious indications that the machine is running in a virtual environment. This also applies for different networking services that run, so a simple search for the string “VMware” in the output of “netstat” would often tell if it is a virtual machine or not. Since virtual machines does not have its own hardware checking the MAC address of the machine could also be an easy way of virtualization detection. Virtual machines would be provided a MAC address starting with “00:0C:29”. A commonly used method including the usage of a virtual machine today is automatic malware detection with sandboxes, these applications are running different malware samples in a virtual environment and log the behavioral. For a malware to detect if it is executed in such an application, it is common to test for specific sandbox or virtual machine artifacts. This could, for example, be the position of the mouse pointer which would in a newly started machine be exact at the middle of the screen, and it would not move at all. This along with other typical unmanned artifact could be used to detect if the malware is executed in a sandbox.

Common malware file types
Two malware file types are presented to understand how these files are built and how they can be analyzed.

2.1.8 PE files

The portable executable(PE) format was introduced by Microsoft in the 1980s. It was meant to be a file format for most types of systems, not just Intel and Microsoft [52]. The PE format originates from the Common File Format (COFF), and the name, portable executable, was chosen because its intended to be the common file format for all types of CPUs and operating systems. The format has been quite unchanged through its lifetime, there have only been minor changes when Microsoft introduced the 64-bit version of Windows. There were no need to add new fields, and only one was removed. The main difference is that the genuine fields were extended from 32-bits to 64-bits. So, to a large extent, they have succeeded, all windows distributions can read the same format, the PE [53]. Even though the file format has remained nearly unchanged for decades, not all fields are there for only practical reasons. The PE file format has some "magic bytes" just like any other well working file format. The two first bytes of a PE file is "4D 5A", which is the letters "MZ" converted to ASCII. This happens to be the initials to one of the MSDOS developers named Mark Zbikownski. Besides from this, there is only one other field that determines if it is a PE file or not at the start of the file, and that is the field "elfanew", addressed at 0x30. This address points to the value where the extended file header starts, which specifies if it is a PE file or not, by the bytes "50 45" or "PE" in ASCII. From here the PE file structure begins, and this will be presented further in this thesis. Another artifact worth mentioning with the PE file is that the data structure of a PE file loaded from disk is equally loaded into the memory. This means that if you know how to handle a PE file on disk, you can expect to find the same information in the corresponding position in memory after it is loaded [54].

Structure

The PE file structure will be presented to have a uniform comprehension of how a PE file is built. The overall file structure is shown in Figure 11, where there are to main components, the header-part, and the section-part, which both is parted into several different portions. As a standard, the structure must contain a DOS-header, DOS-stub, PE-file-header, Optional-header and a section table which is an array of section headers followed by the belonging PE file sections.

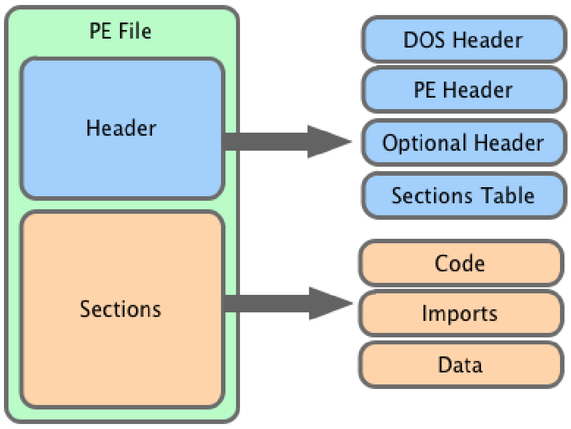


Figure 11: Visualization of the PE file structure, [5]

Header

The header consists of a DOS header as mentioned earlier with "magic bytes" and a Microsoft MS-DOS stub which is the following text string "This program cannot be run in MS-DOS mode.". Followed by the PE signature presented by the bytes "50 45" or "PE" in ASCII, an optional header immediately followed by the section table [55].

PE Header

Like other file headers, the PE file header is where the file have gathered information about the file stored at a defined location for a file parser to read. This will give a good impression of what the file will look like and how it should be handled. Some of the information in the PE header is the size and location of data areas, code, initial stack size, and intended operating system. All this is referred to in different fixed positions in the form of a WORD or a DWORD(double), depending on the different information that is relevant for the various fields. The number of different fields is too big to describe every single one of them, but an example of a field and different values it can obtain will be presented. All fields in the header will be presented with a fixed start-reference "IMAGE/FILE/-HEADER" then the field will be defined by a field name. An example could be the "Machine" field, which is a WORD(16 bits), this position in the file can contain different values which refers to what kind of CPU the file is intended for, some examples is shown in Table 1 [53].

Table 1: IMAGE
FILE_HEADER machine field values

Hex Value	Refered Value
0x14d	Intel i860
0x14c	Intel I386 (same ID used for 486 and 586)
0x162	MIPS R3000
0x166	MIPS R4000
0x183	DEC Alpha AXP

As mentioned the PE file header contains different fields that represent artifacts about the file. The header contains the following fields: Machine, NumberOfSections, TimeDateStamp, PointerToSymbolTable, NumberOfSymbols, SizeOfOptionalHeader, Characteristics.

PE optional Header

This section is not as optional as look from the name for a PE file to work properly. There are some fields, arranged in the same way as the PE header, that is very important for the PE file to run. Like the "AddressOfEntryPoint", which point out the address where the execution of the file will begin. There are a lot of different fields(40) that may be filled in. Describing characteristics like the different sections sizes, where data starts and which interface the PE is designed for.

PE Section table

Within the PE file, there will be some sections, these sections are labeled in the headers in the section table. This table will describe the sections with names, the file size of the section and virtual size, location, flags associated with the section and location of the section [53]. Simplified, this section is used to describe the following sections in the file.

Sections

The sections of the PE file is where the content of the file is stored. This is where code, data, resourced and other executable information is stored. Every section has an own header that follows a standard structure and a place for raw-data, this part is not that defined or organized in any way. It is possible to name the sections whatever the author want to, but there are some standard names that should be used when creating a PE file. Moreover, these different sections store different information with different properties. When a program is compiled the code and data is split into different sections and further on when the program is executed is it loaded into different base addresses within the program image, therefore it is important that the different sections are presented comprehensibly. Some of the standard names and different properties of each section are presented in Table 2.

Table 1: Some standard section names and properties of the sections [56]

Section Name	Description of section
.text	Main program code- usually execute and read access only
.data/.code	Main initialised data code that is used by the program.
.rsrc	Contains the Windows Resource data.
.reloc	Base relocations.
.idata	Imported function data.
.rdata	Read only data.
.tls	Thread Local Storage. Data private to each thread
.debug	Debug information.

2.1.9 Executables and DLL's

Since executables and Dynamic Linked Libraries(DLL) is the most used types of PE files and especially when it comes to malware, a short description of the difference is provided. Normally the difference between these file types is that a DLL is a file that offers other functionality to other instances in the form of code. Programs or executable files can imports DLL's to add functionality the program. DLL's are the current Windows way to use libraries to share code among multiple programs. So, an executable is a standalone file able to run by itself but uses DLL's to perform imports of different code that it uses. The file characteristics are very alike, but there is one field in the PE file header that specifies what it is. The "Characteristics" fields specify this, this field could be three different values 0x1, 0x2 or 0x2000. Where 0x2 notes that it is an executable and 0x2000 that it is a DLL [53].

2.1.10 ELF files

ELF is short for Executable and Linkable Format (ELF) and was earlier called Extensible Linking Format. ELF is a standard file format for executable files which includes program files, object files, core dumps and shared libraries [7]. The file format was published in in the late 1990's and where quickly adapted by the Unix systems as the standard executable file format. In 1999 it officially became the standard binary file format for Unix system. One of the major reasons it was accepted as the standard file format so early was because it is cross platform, extensible and not bounded to a specific type of CPU architecture [8]. In other words, is was very flexible.

The ELF file is recognized by a set of magic bytes at the start of the file, this is 0x7F followed by 0x45 0x4c 0x46, or "ELF" in ASCII, these four bytes constitute the magic number. This file start is mandatory, and the rest of the file consists of a header, thereby followed by a program header table, section table or both [57]. After this, the sections of the file containing data in the form of code, text, and resources follows, not very different than the structure of a PE32 file. The different headers and section will be further described to understand the file format in depth.

File layout and structure

The total structure of a ELF file is a header followed by data which can include a programmable header table, as section header and data referred to in the headers. A simplified visualization of the ELF structure is presented in Figure 12.

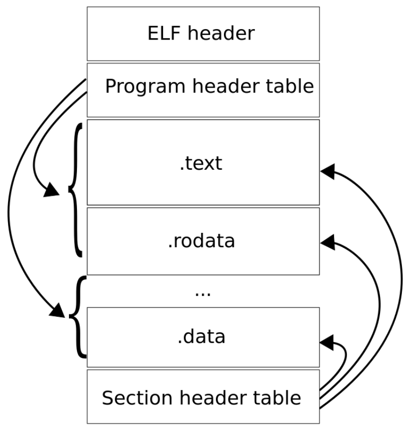


Figure 12: Visualization of the ELF file structure, [5]

Figure 12 also presents how the header tables and section header table referees to the data inside the "data" section of the file.

Header

The ELF header is used to define some parameters about the file such as if the file is meant for a 32-bit or 64-bit architecture, which operating system it is meant for and processor specific options to mention some. In the table 1 the structure of the different header fields with a description is presented. Table 1 is for a 32-

Table 1: ELF header with description of the fields [7]

Header field	Description
e_ident	Specifies to interpret the file, independent of the processor contents
e_type	This member of the structure identifies the object file type
e_machine	Specifies the required architecture for an individual file
e_version	TIdentifies the file version
e_entry	Gives the virtual address to which the system first transfers control
e_phoff	Holds the program header table's file offset in bytes
e_shoff	Holds the section header table's file offset in bytes
e_flags	Holds processor-specific flags associated with the file.
e_ehsize	Holds the ELF header's size in byte

bit ELF file, the structure is almost identical for a 64-bit ELF file except for the length of the fields.

Sections

The sections of a ELF file is used to divide the file into an orderly format for both the program developer, but also the program so link what content is found where. As mentioned in table 1 a ELF file can hold a program header, and a section header, each of these are section has its header as referred to in the ELF main header in the field "e_phoff" and "e_shoff" [57]. This offsets point to the section header for the oncoming section. This header is similar to the main header and has the main purpose of pointing to where the section is stored, what it contains and how it should be loaded when the file is executed. Exactly like the PE32 sections the name of the section often described what the section contains, even though the name of the section can be determined by the programmer. Nevertheless, there are some standard naming conventions which are known as "special section" which has specific types and functionality if the names are used. An un-exhaustive list of special sections names are provided in Table 2 with a description of what the main functionality is.

Table 2: ELF special section description [8]

Section name	Description
.data	These sections hold initialized data that contribute to the program’s memory image.
.debug	This section holds information for symbolic debugging. The contents are unspecified. All section names with the prefix .debug are reserved for future use in the ABI.
.dynamic	This section holds dynamic linking information. The section’s attributes will include the SHF_ALLOC bit. Whether the SHF_WRITE bit is set is processor specific. See Chapter 5 for more information.
.init	This section holds executable instructions that contribute to the process initialization code. When a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called main for C programs).
.rodata	These sections hold read-only data that typically contribute to a non-writable segment in the process image.
.tdata	This section holds initialized thread-local data that contributes to the program’s memory image. A copy of its contents is instantiated by the system for each new execution flow.
.text	This section holds the “text,” or executable instructions, of a program.

These sections are recognized with the prefix "." before the name, these names are reserved for the system although applications may use this naming if the implementation is according the documentation [8].

3 Related work

In this section provides a summary of the work previously conducted on the domain.

Malware analysis as a research field is a well known one. It has been written research and scientific papers about malware detection, analysis and classification for decades, but it is still an equally relevant subject. Even though there are a lot of research, it is not likely that the malware research has reached its final stage, not even close. One of the main reasons for this is that as security experts and researchers are finding new approaches to detect and classify malware, malware writers are finding new ways to avoid these detection mechanisms. And since computer systems are developing dynamically, and new systems with different usage, and miss-usage, are presented every day this subject will be relevant for a very extended period of time.

The complexity of modern malware makes the classification of malware harder [58], an example of this is all the different variants that may be found of one single malware. The Agobot [59] has been seen in nearly 600 variants, with versions containing different features, making it even harder to link the samples together.

One of the biggest reasons for variations of malware samples is when security experts created content based signatures to detect malware, and malware writers then started applying self altering malware. This automated the creating of malware with slightly different footprints to avoid the content based signature detection. This alongside that signature based detection often focus on specific behavior from an exploit, which often varies from each system it is specified to exploit, which in turn causes even more needed signatures. Since it is practically impossible to create content based signatures for all types of malware and exploits, intrusion detection systems and anti virus would be inconsistent across products, incomplete across malware and fail to be concise in their semantics [58].

This forces the defending security experts to find alternate ways to automatically detect and classify malware, to detect and manage new threats. This challenge introduced the dynamic analysis of malware which in some ways are much more profound and often results in much more data to analyze. Hence more resources are needed both to gather the information and to analyze it. In many cases, the amount of information available to gather from a dynamic analysis is extensive therefor some artifacts of attributes is chosen to investigate in depth. To address the problem [58] proposed a classification technique where the behavioral of the malware was described by monitoring the changes in the state of the system, in other words, processes created and files were written. The approach used was in contrast to this thesis for not, focused on individual system calls, but focused on "what the malware does on the system [58]".

The conducted project in [58] justifies this statement by claiming that to evaluate individual system calls may at such a "low level" that is do not provide meaningful information, therefore they addressed the problem at a higher abstract layer. Other research has used a lower lever for behavioral detection [60, 61], these projects focused on detecting patterns that were common in several malware samples to detect how they behaved. They addressed this by looking at typical semantic routines malware performed when searching for email addresses or typical deobfuscating loops.

3.1 Call-graphs used for malware detection and classification

Most research address the problem of detecting if a sample is malicious or benign, [13, 9, 62] is examples for this were different artifacts and approaches to detection is performed. Joris and Sand [13, 9] both utilize call graphs to detect and classify malware respectively. Sand [9], which is a previous master thesis from NTNU in Gjøvik, primarily focused on the applicability of information based dependency matching, and if this was useful against obfuscation. It also points out which features that has the best detection rate for the different layers of call graphs, either system, library or hybrid calls. Research focusing on call graphs will be discussed further, but first related work addressing malware classification will be presented.
Applying call graphs to detect or classify malware has been conducted in previous research. Whether call graphs are extracted statically or dynamically from the binary, there are mainly three different approaches. Although they are similar, they have distinct differences, these will be more discussed in chapter 5.

- **GED** - Finding graph edit distances
- **MCS** - Finding maximum common subgraph
- **Vertex/Nodes** - Matching nodes and edges in a graph structure based on the relations

Younghee et al. [63] applied call graphs extracted dynamically to search for maximal common subgraphs. A behavior graph was obtained by capturing exclusively system calls during execution of samples and then compared to other graphs. The comparison was metric, and the similarity was the number of nodes from the maximal common subgraph divided by the total number of nodes in the entire graph. This classification was tested against a set of 300 malware assigned to 6 families and resulted in a fast classification approach with a low false positive rate when applied to benign software.

Ero Carrera and Gergely Erdélyi, through F-Secure, released an article of utilizing call graphs to match malware [64]. The call graphs were derived through

Ida Pro [65], where they also introduced the Ida Python API [66]. The graph matching was conducted through creating an adjacency matrix for each graph. This matrix will have a row and column for each of the functions presented in the binary, and its element in position (I, J) will indicate whether a function in row I performs a call to the function in column J [64].

Another duo that utilized the adjacency matrix to describe a call graph is Briones and Gomez [67]. They built their research upon the results of Carrera and Erdélyi [64] and added a signature from the opcodes used in the function matching. This signature was a result of the number of blocks in the function. This was then used for exact matching of sub-functions.

Malware clustering based on call graphs was conducted by Kinable [13], he tested two different algorithms to find graph edit distance approximations, genetic search, and bipartite graph matching. Genetic search provided slightly better results, but considering the resource consumptions bipartite graph matching was evaluated better. Kibable where provided with unpacked malware from F-Secure and did not have any goal of fighting encrypted or obfuscated malware. Hence he did conduct a static analysis. After deriving a graph edit distance approximations, Kinable applied three clustering algorithms to look for patterns. The clustering was applied to a set of real malware samples, and the result was evaluated against manual classification provided by F-Secure. The results of the clustering point out that is was possible to classify malware families by using the DBSCAN clustering algorithm accurately.

Lee et al. [68] also conducted static call graph extraction to detect metamorphic malware. They converted the API call sequence into a call graph to describe the semantic of the malware. The extracted call graph was then reduced to a code graph to be matched with predefined signatures. The detection was tested against a real world malware sample set and had a 91% detection rate.

An attempt to detect obfuscated malware using function call graphs was conducted bu Xu et al. [69]. Xu utilized the caller to callee relationships, where the opcodes were describing the functions. The matching was related to the graph edges between functions, and graph coloring techniques were used to measure the similarity metric between two function call graphs.

Other related work is [70, 71] where [71] proposed an automatic extracting technique for behavioral specifications related to graph mining and concept analysis. Which turned out to generate a very low false positive rate when used to detect malware. In [70] Babic et al. tried to generalize malware behavioral from observing tree automated interference in dependency graphs.

3.1.1 Call graph detection for different layers

L.A. Sand [9] used call graph for malware detection and described a hypothesis about which layer detection would be most accurate. Sands hypothesis was that it would be easier at a higher layer (library) then for a lover layer (system).

The background for this hypothesis stem from that system calls handles system critical resources, the fact that a number of executions is different regarding a number of calls and that there exists a lot more library calls then system calls. Sand also describes the reason for this as "*The reason that a number of existing calls is different is that the hierarchy of libraries works like a funnel. High layer libraries utilize lower level libraries. As such, there exist a lot of high-level libraries, while the lower one gets, the fewer libraries there is. [9]*". This would also lead to less variety in the system layer because multiple library calls may end up calling the same system calls.

3.2 Information-based dependency matching

To understand information-based dependency matching and how it can be used to detect or classify malware some terminology and approaches to the method will be explained. To extract information to analyze from calls graphs, it is crucial that information gathered from a sample is relevant. Two main methods to do this is either sequential or nonsequential function call analysis. This is basically if the analysis is taking into consideration the order of the calls to look for dependencies [72]. Another approach to information gathering is to see which parameters that are passed to the function when it is called, this may increase the accuracy of predicting what the function call is meant to perform. An example of where both these approaches are used is in [72]. This algorithm applied both the sequence of the calls and the arguments given to the function call, to result in an inter dependencies representing each sample. This is a dynamic analysis technique and executes each sample to collect the traces.

3.2.1 Reliability of information based dependency matching

A method to test the reliability of a method is to utilize synthesize testing, this means to crate tested based on known input data to explore if the results is as expected. Testing related to function call information based dependency matching was to some extent performed in [72, 73, 74], but more thoroughly tested by Sand in [9].

Synthesize testing is described as "*testing 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 I/O, 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 [9].*"

Previous work for graph extraction

Previous work have focused on testing def-use dependencies and obfuscation resilience. This section presents def-use dependencies and obfuscation resilience previously tested.

Def-use dependencies
A def-use dependency is "*express that a value output by one system call is used as input to another system call [72]*". This means how to handle values passed in a function and what they will return to see if they are usable for building the desired graph. L.A Sand [9] Performing this test an assessment of what type of values to test must be done, the tested types was boolean, characters, strings and integers. And the result of this was a set if rules defining the graph extraction.

- **Characters:** no single characters should be used to find dependencies between nodes
- **Integers/Boolean:** no integers/boolean should be used to find dependencies between nodes
- **Strings:** only strings in the form of memory values should be used to find dependencies between

Obfuscation resilience
The other main property tested by Sand [9] was obfuscation resilience. This was conducted because the information collected by samples that are executed is more likely to give a steady dataset then un-executed samples. This is not directly linked to information dependency matching, but is an additional feature when samples is executed. Sand [9] tested for the reliability when code has been altered with dead code insertion, code substitution, string obfuscation and when samples are packed.

The results was in general positive, and a total of 5 out of 11 methods showed that it did not affect the graph extraction [9]. A comment to these results is that obfuscation may intervene for different analysis, malware can have static analysis obfuscation or dynamic analysis obfuscation or both. This means that it would be preferable to test this with a known malware set with known obfuscation techniques both static and dynamic.

3.3 Fuzzy hashing used for malware classification

A quite different form of malware classification and primarily with respect to classifying malware into families are to apply fuzzy hash algorithms, also known as fuzzy hashing. This is "a type of compression functions for computing the similarity between individual digital files [75]". The cornerstone of this analysis is to part a file into several small pieces and hash each piece to create a total overview. This to avoid so detection mechanisms would fail if just one bit of malware is altered, which will alter the total hash sum for a file when using, e.g., md5 [76] or sha256 [77]. This approach has been used by quite recognized organizations as Shadowserver [78] and VirusTotal [79], latter developed the application "ssdeep [80]" which is commonly used for this purpose. This is research that is related to this project because it attempts to detect malware that has the same structure and therefore can be classified into the same family. The application and procedure to implement such a classification system are quite different from what this thesis project will from, so deeper elaboration in the field will not be discussed.

4 Call Graph

This chapter presents an introduction to call graphs by describing their origin and connection to function calls and operating system functionality.

4.1 Introduction to Function Calls

Function calls can be parted into three types, library calls, system calls and hybrid calls [81]. These calls are handled by the operating system which has the primary task of keeping track of and allocate computational resources. To explain what a function call is a step backward is appropriate. Function calls are created in computer programs and are an implementation to structure and reuse code in these programs. When a program is executed it loads into memory and initialize the memory space needed to execute the program, when this is done different sections of the program is loaded in its representative memory space, and all functions are placed in memory. When a function is called, an address in memory is jumped to, and the code there is executed. As mentioned, different calls are available on a typical computer today, the partition used in this thesis is based on which context the call is executed in [81]. The term used to divide these layers are user and kernel-mode. The processor executing a program is changing between these modes depending on the type of code the processor is executing [6]. And as a thumb rule, core components related to the operating system runs in kernel mode and application runs in user mode. An example of this is that a program can be launched in user mode, but need to access kernel mode when for example file write or read operations are needed. In this case, the application needs to access the disk by utilizing system calls such as "WriteFile" in "kernel32.dll" for a Windows system [82]. An overview of the user mode and kernel mode is presented in Figure 13, with the intermediate function calls to traverse from user mode to kernel mode. Basically, a library call is in user mode and calls execution of a user mode function, and a system call is used for code execution in kernel mode. Whereas hybrid calls will be a term used where there are accessing both user and kernel-mode.

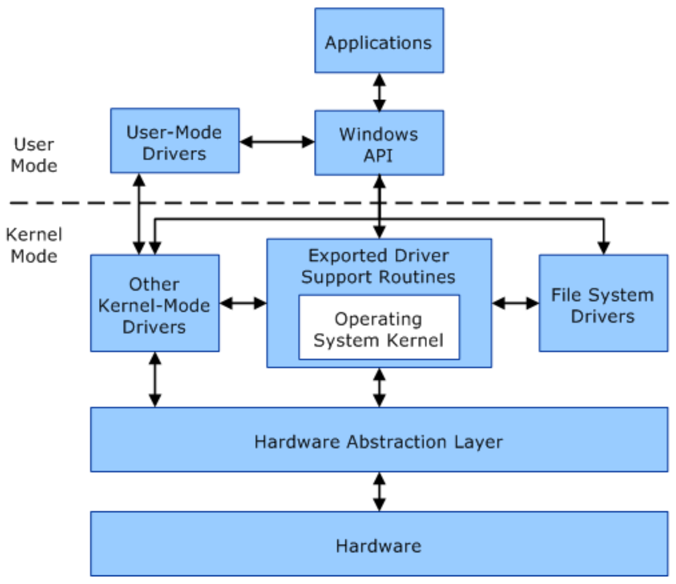


Figure 13: Figure illustrates communication between user-mode and kernel-mode components [6]

4.2 Introduction to Call Graphs

The call graph can simply be described as a summary of the relations between functions in a program visualized in a graph. A call graph is a sequence of function calls described by graph notation where the order of calls is relevant. A graph consists of vertexes/nodes (V) and edges (E). *A call graph is a directed graph whose vertices, representing the functions a program is composed of, are interconnected through directed edges which symbolize function calls [83].*

To understand this, graphs must be understood. Graphs is a method for visual representation and structuring of data. To be familiar with the consent of graphs definitions of graphs, subgraph and graph isomorphism is presented.

Definition 1 *(Graph): A graph G consists of a vertex set V(G) and an edge set E(G). Edge set E(G) is a relation between vertices, i.e. for all (u, v) ∈E(G) if and only if u, v ∈ V(G). For undirected graphs the relations are symmetrical: (u, v) ∈ E(G) if and only if (v, u) ∈ E(G). Directed graphs do not have this restriction. [84].*

Definition 2 *(Subgraph): Graph H is said to be a subgraph of graph G if and only if $|V(H)| \leq |V(G)|$ and there exists an injective function $f : V(H) \rightarrow V(G)$, such that for all (u, v) ∈ E(H) if and only if (f(u), f(v)) ∈ E(G).[84]*

Definition 3 *(Graph isomorphism): A graph H is isomorphic with a graph G if and only if $|V(H)| = |V(G)|$ and there exists a bijective function $f : V(H) \rightarrow V(G)$, such that ∀(u, v) ∈ E(H) if and only if (f(u), f(v)) ∈ E(G), i.e. both graphs are each others subgraphs.[84]*

The simplest form of a graph is two vertex with an edge between. This is an undirected graph with no labels. A visual representation of such graph is presented in figure 14. To enreich a graph, direction can be added. Graph vertices

Figure 14: Visulization of a simple graph

are connected by edges, where the edges can be associated with a direction. A directed graph is formally explained as a ordered pair G = (V,A) where V is vertices and A is a ordered pair of vertices and how they are related.

Another feature of graphing is labeling. A label is a set of symbols, typical labels are integers, strings, colors or any marking. Assigning labels to graphs has the property of creating unique vertices and edges, which furthermore can describe the difference between sections in a graph. Labels can be added to both vertexes and edges. In figure 15 a directed graph with labels for each vertex and edge is presented.

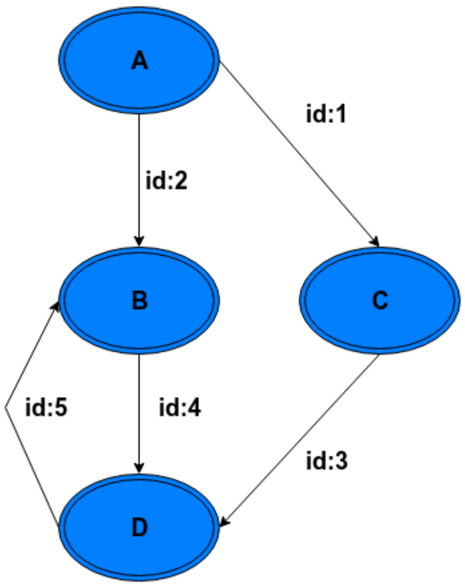


Figure 15: Visulization of an enriched graph

In general disassemblers as Ida Pro [65] labels vertexes based on if the function is a known function call or not. Known functions are named and unknown given a random tracking number. A vertex can thereby be one of the following types:

1. External functions: system and library calls.

2. Local functions, implemented by the program designer.

The main difference here is that system or library calls as mentioned in section 4.1 is stored as a part of the operating system such as Microsoft Windows dll's library, unlike local functions which are created by the programmer. Moreover, there is a logical link between the two types; a local function can call an external function, but never visa versa.

As mentioned a call graph is a sequence of function calls described by graph notation, that in total represents a program. The graph is both directed and labeled, it presents the caller and callee connection of functions in the program. In the same manner as described in section 2.1.4 call graph analysis can be conducted statically and dynamic. If a statical analysis is performed the call graph is built by disassembling the program with a disassembler, and the program is not executed, unlike dynamic analysis where the call graph is derived in runtime by tracing the programs behavior.

The information available in the call graph for each method varies. When using static analysis to collect the call graph, information about all of the functions and function calls is presented in the graph. This approach may lead to an overwhelming number of vertexes(function calls) when big libraries are imported, but not called. Also by utilizing static analysis, the actually executed flow of function calls is difficult to anticipate because the program can take different paths along the execution flow.

This is where dynamic analysis has an advantage, this approach tracks only the executed functions and the relationship between the caller and callee. This, of course, has the disadvantage of not tracking all functions in the binary, hence not covering the total ability of the program.

As indicated, graphs can be built with varying degree of information or precision. The amount of information for each vertex is often related to the context of the vertex. This is a term also used for call graphs, they can be context sensitive or context insensitive. Context insensitivity is when there is no additional information added to a vertex, on the other hand, when a graph is context sensitive additional information is added, such as arguments and return values. Context sensitive call graph gives a more precisely approximation of the behavior of a program, at the cost of time and computational resources. The most precise call graph possible to derive is to add a full call stack to each vertex, this is known as calling context tree, and is very resource demanding. An approach to a context sensitive call graph is to represent each vertex with a context containing the caller arguments and the return value of the called function. This the call graph definition used for this thesis.

Definition 4 *(Context sensitive Call Graph): A call graph g is a directed graph defined by 4 tuples g = (Vg, Eg, Lg,Lg), where Vg is the finite set of vertices, each corresponding to a function; Eg $\subseteq$ Vg $\times$ Vg is the set of directed edges where an edge* *from f1 to f2 implies that f1 contains a function call to f2, but not vice versa; Lg is the set of labels each of which is comprised of 3 elements: symbolic function name, call value and return value; Lg : Vg $\rightarrow$ Lg is the labeling function that assigns labels to vertices. [84].*

5 Graph matching

This chapter discusses graph matching approaches and the challenges of graph matching.

5.1 Graph matching approaches

By utilizing properties of call graphs to either detect or classify malware, a comparison of the extracted properties must be performed to distinguish the difference in samples. This may be done in various ways, but the main objective is to measure how different or similar two graphs are.

Definition 5 *(Graph matching): For two graphs, G and H, of equal order, the graph matching problem is concerned with finding a one-to-one mapping (bijection) $O : V(G) \rightarrow V(H)$ that optimizes a cost function which measures the quality of the mapping [13].*

Bengoetxea [85] divides methods of graphs matching into two approaches; exact graph matching and inexact graph matching. The main difference is that exact matching is exhaustive which means it finds an optimal solution or not a solution at all, while inexact is non-exhaustive. In other words, exact graph matching searches for graphs that are an exact match, while an inexact matching search for graphs that are somehow similar to or alike. This means that for inexact matching to implemented practically, it needs to set some threshold to decide if the graph is similar enough. This can, for example, be implemented by use of Graph Edit Distance (GED), where there will be a maximum edit distance for the graphs to be matched. An overview of the different graph matching methods is presented in figure 16 [85].

5.1.1 Exact Graph matching

Within the term *Exact graph matching* there is two subgroups as visualized in Figure 16. While the method of graph matching seeks to find exact identical graphs, the difference between graph isomorphism and subgraph isomorphism is related to completeness. The graph isomorphism method requires that the whole graph is identical, while the sub-graph isomorphism requires only a part of the graphs to be identical [85]. To relate this to malware detection, full graph isomorphism would be quite useless since it requires an absolute identical layout. Which is not common in general malware and software because of different compilers and changing behavioral at runtime. A definition of graph isomorphism was presented in Section 4.2

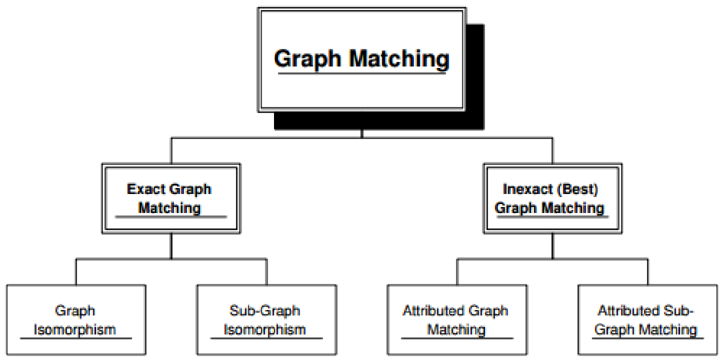

Graph Matching
Exact Graph Matching
Inexact (Best) Graph Matching
Graph Isomorphism
Sub-Graph Isomorphism
Attributed Graph Matching
Attributed Sub-Graph Matching

Figure 16: Graph matching methods

On the other hand to be more unaffected from how alteration of call graphs affect the matching, sub graph isomorphism is more suited. This because this method can find smaller parts of the graphs that can be matched, and not to discard the whole matching if it is not an absolute match [9].

5.1.2 Inexact Graph matching

The term *inexact graph matching* can also be divided into two subgroups. And as Bengoetxea states what excludes this method from isomorphism method is that " *it is not possible to find an isomorphism between the two graphs to be matched. This is the case when the number of vertices is different in both the model and data graphs [85]*". So this method should be utilized when there is a lot of difference in the matching objects. Inexact graph matching is the most suitable method in this project because the difference in call graphs may vary from such a vast amount of variables, different parameters, address values, compilation method and library version. This messy environment results in a situation where exact matching is hard, therefor an approximation must be applied, such as inexact graph matching.

This can be done by detecting the maximum amount of common subgraphs from two graphs (MCS), this is sometimes referred to as Longest Common Subsequence (LCS). The MCS is a technique that tries to find the maximum common subgraph for two graphs as the name intent, this can be done by representing the graph as a searchable string. Then match vertexes from an input graph against a data graph, when there is a match, the following subset of edges and vertexes will be added if the graphs are identical. This will the represent the total subgraph.

Definition 6 *(Maximum common subgraph): Maximum common subgraph between graphs G and H, is the subset of vertices in G, $g \subseteq V(G)$, for which there is a bijection to a subset of vertices in H, $h \subseteq V(H)$, g is isomorphic to h, and $|g|$ is* *maximal.*

Another graph matching approach is to find the minimum editing of graphs to make it represent another, also known as minimum graph edit distance (GED). The distance of this algorithm is calculated by relating an alteration with a cost which is summed up and represent the total cost of editing the graph to represent another. A common operation to apply a cost value to is insertion and deletion of isolated vertexes, insertion and deletion of edges and changing the label of a vertex or edge.

Definition 7 *(Graph edit distance): The graph edit distance is the minimum number of elementary edit operations required to transform a graph G into graph H. A cost is defined for each edit operation, where the total cost to transform G into H equals the edit distance. [13]*

5.1.3 Graph similarity

To be able to compare how similar two graphs are a similarity metric for how alike the graphs appear is necessary. This similarity is a offspring from the graph matching, where the graph match is evaluated against the entirety of the graph. This is done do get a value sometimes referred to as a score of similarity between two graphs [86]. Kinable [13] derived a definition of graph similarity which is cited in definition 8.

Definition 8 *(Graph similarity): The similarity σ(G, H) between two graphs G and H indicates the extent to which graph G resembles graph H and vice versa. The similarity σ(G, H) is a real value on the interval [0,1], where 0 indicates that graphs G and H are identical whereas a value 1 implies that there are no similarities. In addition, the following constraints hold: σ(G, H) = σ(H, G) (symmetry), σ(G, G) = 0, and σ(G, K0) = 1 where K0 is the null graph, G σ= K0.*

6 Methodology for multinominal malware classification

The the main research question in this thesis is to describe a methodology for multinominal malware classification using expert call graph matching. This question is addressed through the this and the next Chapter 7. This chapter presents and discusses the methodology for the experiment. Firstly, a top-down presentation of the methodology for automated malware analysis is described. Secondly, a detailed description of how each step in the multinomial malware classification methodology is discussed. The second, steps include the dataset generation, call graph extraction, analysis problems and matching algorithm.

6.1 Malware analysis methodology

Firstly an overarching method for dynamic malware analysis is described to present the methodology conducted later.

Sikorski [4] defines dynamic analysis as "*any examination performed after executing malware.*" This is a simple and straightforward definition, this definition is the base of the methodology in this thesis with a slight alteration. Moreover, the analysis is conducted both *after* and *under* the execution of a malware sample. Automated analysis (Section 2.1.5) is the cornerstone of the methodology used in this thesis. The approach is modified to focus on automated call graph extraction. This because the approach presented in the automated analysis section is focusing on a fully automated sandbox analysis and is not fully unanimous with the methodology for this thesis. The methodology presented is generic for all malware regardless of file type, operating system or functionality.

First step

Firstly an environment for the malware analysis must be set up. This must be an environment capable of running the incoming malware depending on file type. The environment should be as clean as possible regarding traces of an analysis system, e.g., not running superfluous programs when analyzing the malware.

Second step

The next phase is to load the malware into the analysis environment. This could be done in various ways, but it important to do this safely. For instance a network share between an analysis environment and other machines is not recommended since this could open for an exfiltration way for the malware.

Third step

The third phase is to set up the needed software to be able to conduct the extraction of artifacts from the malware when it is executed. This is pre-execution tasks which orchestrate for information gathering, in the context of function call tracing. This can be implemented tracing software.

Fourth step

This phase is related to the execution of the malware, this phase sounds simple but is critical because this is the core component for deriving the data from the experiment. As mentioned in the previous steps, the environment for the execution of the malware must be adapted to conduct this phase successfully. An important aspect of this phase is how the function call of the malware is gathered, is the malware attached to a debugger or is it executed through another software which traces the calls. Either way, the results of the function call extraction must be reviewed to see if the results are correct.

Fifth step

The next phase is to perform post execution tasks, this includes collecting logs or other artifacts the execution of the malware has created. In the context of function call, this would be the output form the call tracing program.

Sixth step

The sixth phase is the cleanup phase. This must be conducted to make the environment ready for a new malware to be analyzed. There is not difference when it comes to function calls compared to other types of dynamic malware analysis, but it is a very important phase since malware can be interfering with each other if they utilize the same kind of system utilities. This can be conducted by reverting the analysis environment to the original state presented in the first step.

For dynamic malware analysis, these steps must be seen in collaboration where each step is conducted in chronological order from step one to six. Figure 17 shows the step used in the methodology in this thesis.

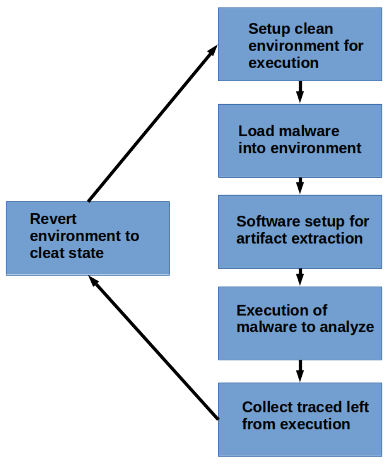

Setup clean environment for execution
Load malware into environment
Software setup for artifact extraction
Execution of malware to analyze
Collect traced left from execution
Revert environment to cleat state

Figure 17: Visualization of data generation from dataset

6.2 Dataset

Research is a result of processed data. A dataset is passed to a model which evaluates or extracts parameters from the data, and the model results are presented. The results are then evaluated, this is the scientific workflow [87]. In malware research a collection of malware is the most common dataset to be evaluated, and a very important property of the dataset is that it applies to the real world i.e. real world samples.

Malware research is not a new domain of research, therefor there exist datasets of previously conducted research. Some of the well used datasets are evaluated to see if they are applicable for this thesis.

6.2.1 Evaluation of existing datasets

This Section explores existing dataset to see how well they apply to the methodology in this thesis. A dataset for this thesis should fulfill some general criteria, this is that the dataset is time relevant, big enough to be used in a valid experiment, be executable, cover different malware families and be pre-classified into malware families.

DARPA datasets

DARPA is short for Defense Advanced Research Projects Agency and has in collaboration with Massachusetts Institute of Technology (MIT) gathered three datasets in the late 1990s [88, 89, 90].

- **Pros:** Sorted malware data, reliable and professional source.
- **Cons:** Old datasets, not time relevant, no access to the malware only information about the malware.

CCC DATAset 2009

CCC dataset was an initiative for anti-malware research, and the collection was conducted by the use of a honeypot operation [91]. The research was published in 2009.

- **Pros:** Sorted malware data, reliable and professional source.
- **Cons:** Short term collection, difficult to gather a broad specter of malware during a time limited honeypot operation.

MWS 2009

This dataset was established in a anti-malware workshop in 2009 at the Toyama International Conference Center in Japan [92]. This is a subset of CCC DATAset.

- **Pros:** Sorted malware data, reliable and professional source.
- **Cons:** Subset of CCC DATAset, this results in a small but profoundly descriptive dataset.

Microsoft classification challenge BIG 2015

In 2015 Microsoft launched a competition where the goal was to classify malware into families based on file content and characteristics [93]. The dataset for this challenge is large and well documented.

- **Pros:** Sorted attack data, reliable and professional source, a vast amount of samples.
- **Cons:** Not executable samples to download, each sample had been altered not to be runnable.

After searching for a suitable dataset for this thesis, there occurred some relevant datasets, but none that could easily be adapted the specific needs to conduct malware call tracing. None of the datasets found, satisfied enough of the criteria to be used. It was therefor decided to collect and generate our dataset. This has the drawback, which our results cannot be easily compared with future work, unless they use the same dataset.

6.2.2 Dataset file type evaluation

Malware is crafted for several reasons, but whatever the reason one fundamental decision is made before a malware developed; what system should it effect. The system type varies from nuclear reactors to home computers, but the most common platforms to craft malware for is Windows, Android, OSx and Linux. For this thesis Windows and Linux files have been evaluated.

Windows PE32 files

This file type has been evaluated because of the worldwide spread of the file system. Windows malware is the most common type of malware because of its vast user mass, everything from airport systems to handheld devices. There are big repositories of malware available, good documentation of different malware types and analysis.

Linux ELF files
This file type has been evaluated because it is a common file type for malware targeting for servers. ELF is common for Linux users which are increasingly popular and therefor increasingly relevant. There are known repositories of ELF malware, with good documentation, available.

Chosen file type
This thesis was intended to use PE32 files for the dataset, for reasons described in section 6.2.6. The chosen file type is ELF files. The main reason for this is problems analyzing PE32 files and deriving complete and understandable trace files for each sample. It was not possible to get enough working PE32 samples for this thesis. Technical challenges faces are described in Section 6.2.6. Knowing the popularity of PE32 malware, solving these issues and getting results for PE32 is highly encouraged as future work.

6.2.3 Dataset for this thesis

The malware set used was VirusShare_ELF, which is a packet of malware available. It contains 2778 unclassified samples. The malware samples were gathered from the malware sharing site VirusShare [94]. This site requires a user to get access to samples, this can be obtained by requesting the site owners for access, with a school research justification. From this, a set of 500 malware samples divided into 5 families. The families derived is known by the names Aself, Cornelgen, Madvise, Rst and Tsunami.

6.2.4 Dataset - Classification and filtering

As mentioned the dataset used was unclassified. This thesis focus on analyzing malware families to discover inter family dependencies, because of this an initial classification of the malware had to be conducted. This was achieved by utilizing VirusTotals [79] database of known malware, and the API integrated by the site. When a sample is uploaded to VirusTotal it generates a report related to the submitted sample, this includes static analysis, and historical classification by 50 anti virus companies. To classify the samples downloaded from VirusShare, each of them was submitted to VirusTotal, the returned report was then processed to see which family the 50 different anti virus companies has related the sample too. This was conducted through a python script added in Appendix A. Then the samples was filtered based on file size and the ratio of connections to the same family. By conducting this, the samples could with satisfactory confidence be linked to the respective family.

6.2.5 Dataset - Call graph extraction

By utilizing on a completely new dataset, it is inevitable to not analyze and process the dataset in some manner. In this thesis, the dataset is malware samples that are being executed, and the malware behavior is traced as it executes functions. To generate this dataset an environment for malware execution is needed along with software to trace the function calls during execution. Firstly the environment will be described then the method for function calls extraction will be presented.

Environment

As presented in section 2.1.6 there are several methods for building a secure environment and keeping it secure. From the three mentioned methods for building an environment, two of them is relevant to this thesis because an environment including a controller is too comprehensive for this thesis.

Hardware implementation

The hardware approximation utilized hardware to create an analysis environment, which needs to be reverted to the original state for each time a sample is executed, to exclude traces left from previously executed samples.

- **Pros:** provides a genuine environment for the sample to run in, avoids anti virtualizations functionality implemented in malware.
- **Cons:** the need to install or revert to a clean state for each malware sample it time consuming and difficult to do properly.

Virtualization

Virtualization has been the defacto standard for malware analysis for some time. An analysis environment can be implemented by visualization tools, such as VMware or VirtualBox [95], this is software that is easy to get a hold of and easy to use.

- **Pros:** easy configurable, relatively low resource-intensive, built-in functionality for restoring machines to previous states, scriptable environment.
- **Cons:** malware can be virtual environment aware and alter functionality based on that.

For this thesis a virtual environment is used, this has the benefit of being operating system authentic and resource affordable. For creating the environment VirualBox [95] is used to run a Linux Kubuntu 32-bit operating system to execute the malware samples. VirtualBox can capture snapshots of the guest operating system which is applicable when executing several malware. The con of virtual environments related to virtualization aware malware is manageable since there been an increase of systems implemented in cloud environments. This means that more and more production systems are running in virtual environments, therefore malware writers often do not include this functionality into malware because this would exclude potential infections.

A snapshot of the clean guest operating system will be the base for every malware to be executed. When the malware is executed a tracing software is attached to the process, and tracks function calls during execution, this will then be written to a log file on the host system. The methodology using a virtual machine is enumerated to elaborate how the different functionality is implemented in the methodology described in Section 6.1.

1. Restore clean snapshot of virtual guest machine
2. Start virtual machine
3. Load current malware to be executed
4. Execute malware and trace function calls
5. Extract log to host machine
6. Revert to clean snapshot

Function call graph extraction

Tracing executed programs is not always trivial. Program tracing is the art of hooking onto a program and record or log information about the behavior of the program. A normal utilization of tracing logs is debugging of a program to look for bugs [96], this is not the case in this thesis. In this thesis, tracing is utilized to gather function calls to get a fingerprint of a malware. Some tracing software is evaluated to present pros and cons of using the different approaches.

IDA PRO

IDA pro is a disassembler and a debugger and provides a variety of functionality for getting insight into how executable code function. IDA has been the defacto standard for the analysis of hostile code and vulnerability research [65]. Moreover, has the capability to represent binaries in various layouts, analyze binaries both static and dynamic, and it has built in scriptability.

Because of its wide use, several plugins have been written to perform automated analysis of binaries, and the Ida developers (Hex-rays) hosts an Ida plugin contest each year. In 2015 the Yaniv Balmas won the contest with his plugin, Dynamic IDA Enrichment (DIE) framework [97]. This plugin offers the possibility to enrich a binary analyzed in Ida with runtime values. As Yaniv explains DIE as "*an IDA Python plugin designed to enrich IDA‘s static analysis with dynamic data. This is done using the IDA Debugger API, by placing breakpoints in key locations and saving the current system context once those breakpoints are hit. [98]*"

- **Pros:** scriptability (python and IDA-c), thoroughly documented, community written plugins to trace programs.
- **Cons:** licensed, complex usage of scriptability even with documentation.

Intel vTune

Intel vTune was developed as performance analysis tool, it is provided with a straightforward user interface and is supported. Along with performance profiling, it can trace calls performed by the running program and list these as a table

[99].

- **Pros:** easy to use and install, lots of information about the profiled program down to mnemonic level.
- **Cons:** licensed, nonconfigurable complex to extract call information with relevant information, not intended as a tracing application.

Clone DLL

Clone DLL is a call hooking technique, this method has been used by malware spying in Windows [100]. The method is conducted by looking for which call a program uses and which DLLs the program imports. When a complete list of all of these is obtained, each of the DLLs can be altered to log when the DLL is called and print which of its functions is being called.

- **Pros:** there exists examples for implementing the code to conduct this, gives full control over the imported DLL.
- **Cons:** the complete list of DLLs and functions called is hard to obtain, especially when there exist a big amount of samples with unknown functionality. Each DLL has to be altered with a program stub to perform the logging.

WinDBG

WinDBG is short for Windows Debugging. This is a multipurpose debugger for Windows systems and is developed to search for errors when the Windows operating system is running. WinDBG is used by enabling it on the computer that is being debugged, it automatically sends all debug information to another computer on a COM port. This requires a computer to receive the debugging information and present this. This setup is required because it hooks into the kernel of the debugged machine [**?**]. It is possible to perform this in with two virtual machines.

- **Pros:** functionality to debug both system calls and library calls in the same program, widely used debugger which is thoroughly documented.
- **Cons:** complex environment to set up, complex user interface even with documentation.

Linux Ltrace and Strace

Ltrace and Strace is an abbreviation for respectively library trace and system trace and is programs that simply runs the specified command or program until it exits. It intercepts and records the dynamic library or system calls which are called by the executed process and the signals which are received by that process. It can also intercept and print the system calls executed by the program [101, 102]. These programs are presented as one because they are so closely related. Both programs are a part of the standard Linux program package and can trace ELF files.

- **Pros:** easy use for tracing of both system and library calls, widely used within the Linux community, functionality for call traces with call arguments and return value
- **Cons:** tracing are only available for Linux executable files

In this thesis, both IDA pro and Lrace/Strace was evaluated as good options for tracing samples and extracting call graphs. IDA pro and the plugin DIE can be used to extract function call graphs at runtime. By using this combination each function call context could be enriched with context information, DIE can extract full context tree as described in section 4.2. One of the goals of this thesis was to enrich the call graphs of samples with arguments and return values [103] of the call context, this could be achieved by the integration of DIE. Using IDA pro led to several difficulties further described in 6.2.6 therefor other tracing programs was considered. Datasets for both PE32 files and ELF files was generated and tested.

Ltrace and Strace were the chosen method for this thesis because of the the structure of the trace files and the ease of extracting a readable and understandable traces for the programs. This was a time-resource question, where Ltrace and Strace were more feasible option compared to IDA pro when the time limit is within a master thesis.

In Figure 18 the secure environment for extraction call graphs is presented. This includes the host machine running a Kubuntu guest machine in VirtualBox.

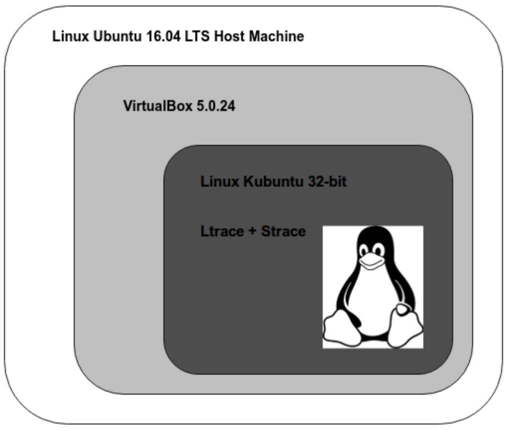


Figure 18: Visualization of call graph extraction

6.2.6 PE32 - Analysis problems

In this thesis, both Linux and Windows samples were tested, and as indicated in some sections there has been some problems with tracing Windows samples. This section presents the main difficulties that were taken into consideration when moving fro Windows samples to Linux.

Tracing difficulties
The initial method chosen in this thesis was IDA pro to trace PE32 samples. IDA was used with a plugin as mentioned in section 6.2.5, along with this plugin scripts to both batch run the files and extract the calls with arguments and return values was written and can be viewed in Appendix C. This was successfully conducted for some of the samples, but had a very low success rate. The environment used for PE32 files was similar to the one finally chosen for ELF samples, but differences when it comes to the guest host. Figure 19 presents the environment with the host machine running a Windows guest machine in VirtualBox. The produced IDA Python script to trace calls and write a graph file is presented in Appendix C.

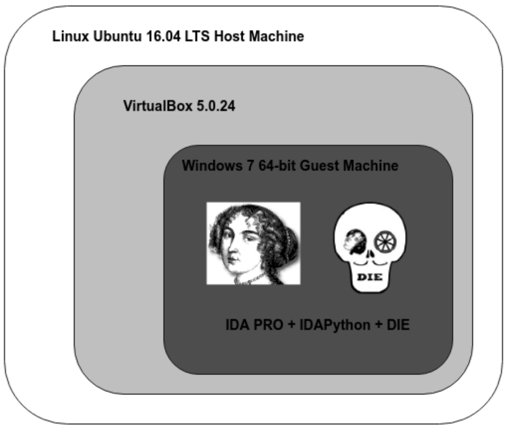

Linux Ubuntu 16.04 LTS Host Machine
VirtualBox 5.0.24
Windows 7 64-bit Guest Machine
DIE
IDA PRO + IDAPython + DIE

Figure 19: Visualization of call graph extraction

The dataset utilized when tracing Windows files was a subset of the dataset used by Shalaginov et.al [104] and consisted of 1000 samples equally divided into 10 families. One of the major difficulties was to get the samples to run smoothly within IDA pro.

When debugging why the samples did not run as intended, a problem with exception handling occurred. Exception handling is a mechanism to handle errors occurring, this is often implemented in programs to handle faults, or in operating system if the program does not take account for the fault. There are several ways of handling exceptions based on why they occur, IDA pro has a built in exception handler to deal with this. Within this handler, there are options to either let the IDA use its handler, pass it to the operating system or to pass the exception to the application running. All three methods were tested, but all options caused the application to crash. For several samples this crash was found place in the unpacking 2.1.2 of the malware, the reason for this could be, but is not limited to, anti debug functionality, exception handling code execution or environmental issues.

Anti debug functionality may be implemented by some malware to prevent analysis as described in Section 2.1.7. In this case this may be done by raising a flag is the malware detects that is has a debugger attached, this is easily implemented by utilizing a Windows library developed to check for this as mentioned in 2.1.7.

Exception handling code execution is a malware code hiding technique [4] used to hide the code executed by the malware. This is conducted by willingly causing an error in the execution of the program which will raise an exception. If the the malware author provides extra code in the handling of the exception through the Structures Exception Handling (SEH), it is possible to hide code from some malware analyzing tools, since they do not always search for code inside these section of the file.

This could also be because of the environment the malware was executed in. Malware, like all other programs, have difficulties running in environments it is not developed to run in, and it is not unlikely that malware authors do not have thoroughly tailored the malware to be able to run in different version of operating systems or that is missing libraries to be able to run.

When analyzing some of the samples that did not execute as expected, the majority of these did not execute because of environmental issues. An example is that a sample crashed if the filename did not match with a the hard coded name in the sample, this could also be an insurance for the author of the malware that it would not be used or analyzed when not ran in the correct setting. This makes it difficult to create an analysis environment for each and every sample.

Because of these problems, only 4% of the samples generated a comprehensive graph of the whole sample when it was executed. Moreover, it would be too time consuming to either manually alter the environment for each sample or to develop an automated environment for handling this.

Overhead in calls - understandability

When working with a dataset, it is crucial to understand the dataset that is handled. As mentioned, some of the samples did execute properly and produced a valid call graph. Another problem occurred when deeper analysis of the call graph produced by the samples was conducted, the graphs was complex to understand.

This was further tested by writing small test programs such as "HelloWorld", the programs executed well, but there was an overlay of calls done before the core of the written program was presented in the call graph. To visualize this two programs written in C is presented with the call graph for each of them.

```
######## HelloWorld.exe ########
#include <stdio.h>
int main()
{
printf("Hello World!\n");
return 0;
}
```

```
######## PrintRandom.exe ########
#include <stdio.h>
#include <stdlib.h>
int main()
{
int x;
x = rand();
printf("%d", x);
return 0;
}
```

Each of these programs is represented in Figure 20 where the calls recognized in the programs such as printf, rand and puts is colored in green and unknown calls is colored in red. This was an attempt to figure out the different overlay which was produced when tracing a small program. A snippet of the picture is presented in Figure 21 to visualize the actually recognized calls.

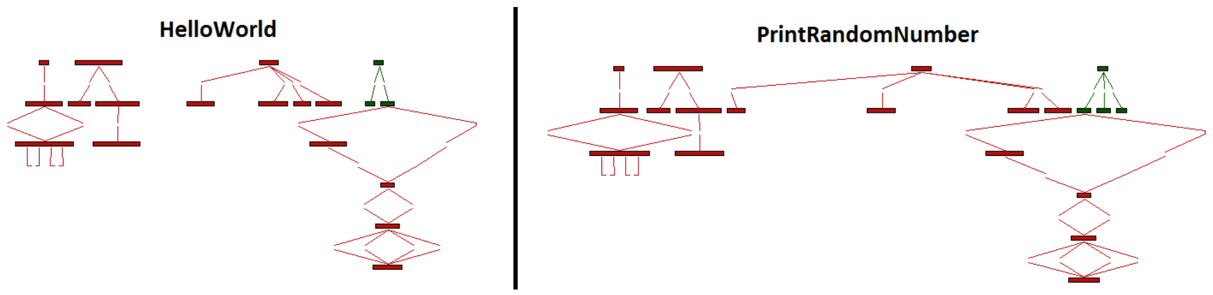


Figure 20: Call graph for both HelloWorld and PrintRandom

An explanation for this is the noise a compiler or loader created when a program is either compiled or executed. In this example the compiler used was MinGW(GCC:(GNU)5.3.1)). Because of the unfamiliarities of the call graphs produced by Windows samples and need to understand the results of the dataset, this thesis does not provide any further analysis of Windows PE32 samples.

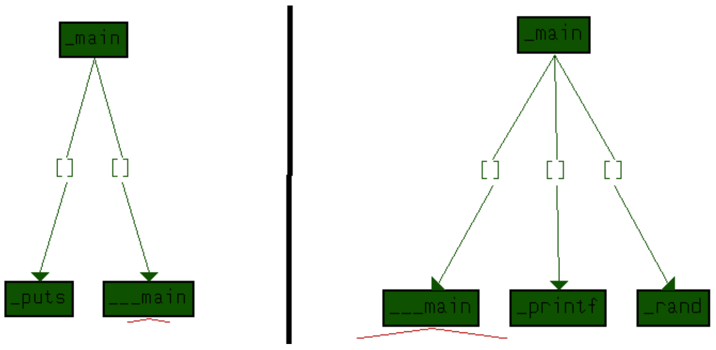


Figure 21: Snippet of figure 20

6.2.7 Dataset statistics - description

Insight and description of a dataset are important to understand what a dataset represent. This section uses statistics of the dataset to describe what is included in the dataset and represents it in a table. The reason for doing this is to look for patterns in the dataset regarding some metrics that may reveal unnoticed structures. The metric used for evaluation in this project is nodes and edges as discussed in the the Call graph section 4.2. Table 3 presents the total number of samples, total number of samples for each family and the sum, mean, max, min and median for all of them. Also, the table divides between system and library layer because one of the research questions addresses this problem specifically.

Table 3: Node(N) and edge(E) description of dataset

Family	Aself		Cornelgen		Madvise		Rst		Tsunami	
	E	N	E	N	E	N	E	N	E	N
Total	1344341	1805476	485088	955523	353459	803270	502205	951121	1654725	2139286
System										
Sum	1211427	1225125	241444	264343	34421	36631	98871	100122	1207296	1224112
Max	42561	47686	12221	15334	43439	56463	14412	145667	87963	87124
Min	4	4	5	4	4	4	8	8	34	34
Mean	100956	102093	16096	17622	1811	1927	5203	5269	18292	18850
Median	161	135	109	111	80	56	72	80	131	144
Mode	161	137	97	114	86	57	72	80	161	140
Library										
Sum	132869	580351	243644	671180	319038	766639	403334	850999	447429	895164
Max	41644	41645	21778	21779	35255	35256	38211	38212	24485	24486
Min	3	3	4	4	4	4	2	2	3	3
Mean	1428	6240	2511	7125	3625	8711	4745	10011	4714	9422
Median	7	8	5	6	5	6	6	6	60	61
Mode	5	6	6	7	5	6	1	2	52	53

System versus Library layer

Table 3 covers statistics about the dataset divided into system and library layer, and there are some artifacts revealed by this statistics. In general system calls have more nodes and edges then library calls, this is not surprising since one library call may be used to call underlying system calls. This will also be a factor when system and library layer is used together in classification if this is not accounted for. The distribution between the families is separating the families to some extent, the Aself and Tsunami families stands out with more nodes and edges the other families especially at the system layer. This indicated that there are more chances to detect nodes and edges for these families and this can be an advantage when it comes to classification.

6.3 Classification algorithm

Matching graphs is complex, and there are different algorithms for matchings graphs based on what data the graph is composed of. As mentioned in section 4.2 the degree of precision in the graph is determine which information that is included to describe the graph. In this thesis, a quite precise graph format has been chosen as described in Section 4, which builds up a complex graph. One approach to overcome the complexity problem with graph matching is to simplify the graph, but still, keep the accuracy of the graph. The classification algorithm should take this into account, an inexact graph matching can be utilized for this purpose. Therefore an attributed sub graph matching will be used in this thesis.

Graph matching implies that one graph is matched with other graphs, this lies in the nature of matching. In this thesis, expert knowledge is utilized as a part of the matching. Graphs formed to represent the different families will be marked as an expert graph which each unknown sample will be compared against. By doing this, a match rate for how similar a graph is to the expert graph can be found for each unknown sample. The deriving of these graphs is based on analysis of the different families and foreknowledge of what to expect from the type om malware, this will be further discussed in the experiment Chapter 7. Since expert knowledge is used in this thesis, a short introduction is provided.

What is expert knowledge

Expert knowledge is, as the name intends, knowledge that is gathered from experts in a specific field. There are several definitions of expert knowledge; Booker defines expert knowledge as "what qualified individuals know as a result of their technical practices, training, and experience [105]". While Perera discusses how experts are identified by their training, experience, qualifications and public recognition [106]. These are only a few of many discussions for what is needed for someone to possess expert knowledge. Even though there are some differences is what is included, there are many commonalities between them. An example of this could be an expert in landscape ecology who has education, years of practice and who can solve complex problems based on the knowledge and analysis of the specific problem.

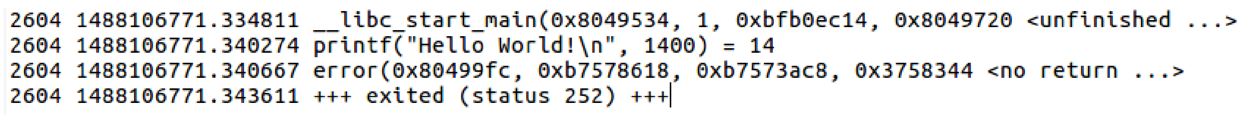
Expert knowledge may be seen as the ability to perceive systems and how they can organize and interpret the information provided. As Ericsson et. al [107] points out, it is important for an expert to analyze and interpret the information served from a system for the expert to make pensive decisions.

6.3.1 Matching algorithm - Node and Edge cover

The matching algorithm in this thesis is as mentioned an inexact attributed graph matching. To be able to use this, an approximation must be done to convert the complexity of the graph into a matchable format, therefore this section presents samples of the dataset to presents the final data matched and also a description of the exact algorithm used in this thesis.

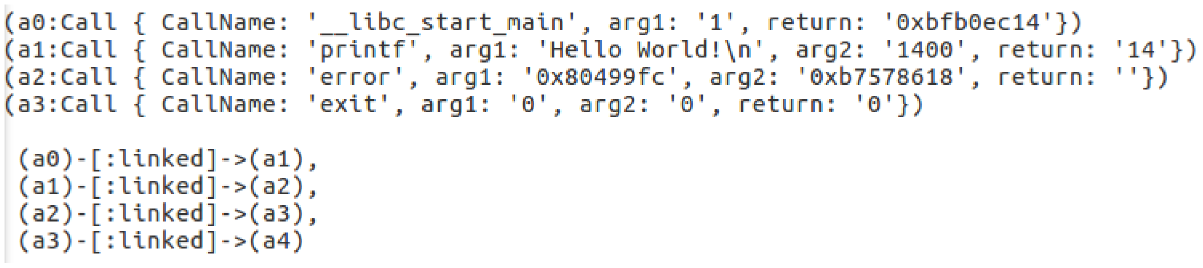
The graph format is originally derived from a Ltrace or Strace output, this output is then pre processed, so it is readable in a graph notation language. The graph notation language in this case is neo4j [14], this was used because of the capabilities of visualization and graph querying. This will be further discussed in section 7.

Figure 22 represents a sample of the output from Ltrace. This output is then processed to become readable for Neo4j, Figure 23 presents the result of this conversion. This is the same trace. Finally, a visualization of the graph in Neo4j, this is all from the same trace, which it the trace from the C-program "Hello World" listed in section 6.2.6



```
2604 1488106771.334811 __libc_start_main(0x8049534, 1, 0xbfb0ec14, 0x8049720 <unfinished ...>
2604 1488106771.340274 printf("Hello World!\n", 1400) = 14
2604 1488106771.340667 error(0x80499fc, 0xb7578618, 0xb7573ac8, 0x3758344 <no return ...>
2604 1488106771.343611 +++ exited (status 252) +++
```

Figure 22: Call graph in string format



```
(a0:Call { CallName: '__libc_start_main', arg1: '1', return: '0xbfb0ec14'})
(a1:Call { CallName: 'printf', arg1: 'Hello World!\n', arg2: '1400', return: '14'})
(a2:Call { CallName: 'error', arg1: '0x80499fc', arg2: '0xb7578618', return: ''})
(a3:Call { CallName: 'exit', arg1: '0', arg2: '0', return: '0'})

 (a0)-[:linked]->(a1),
 (a1)-[:linked]->(a2),
 (a2)-[:linked]->(a3),
 (a3)-[:linked]->(a4)
```

Figure 23: Call graph in graph format

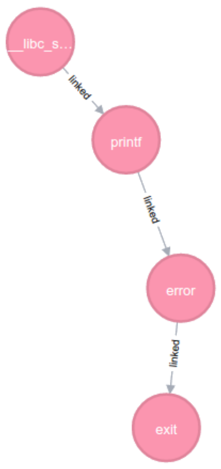


Figure 24: Graph visualization of the mentioned formats with 23 as input file in neo4j

The matching algorithm used is based on plain text matching, this, because it is easy to implement and since the notation of a file, is manageable regarding file size. Matching scripts are presented in Appendix B. To shortly describe this the matching consist of five steps.

1. load sample and expert graph
2. match each node in sample for existence in expert graph
3. match each edge in sample for existence in expert graph
4. weight the matches
5. calculate match ratio

The principle behind the matching is to look for vertex(node) and edge cover. This is a mathematical discipline in graph theory [108] which applies to both nodes and edges. It is mostly used to find a subset in graphs where the minimum size node cover represented in both graphs [109]. In this thesis, an approximation of this is used to look for node and edge cover for each sample matched against the derived expert graphs.

The cover algorithm [108] is used to measure the similarity between two strings by finding each node or edge in the graph matched. Since the graph is preprocessed into a string, the algorithm is able to match strings. To match a call graphs, the proposed algorithm allows mapping of a path (i.e., a path or an edge) from the query graph with a path (i.e., a path or an edge) in the graph using the cover algorithm [110]. These two paths must have the same labels for all edges in the path. The proposed algorithm, as shown in algorithm 1.

To present a similarity *sim(H,G)* between two graphs the following algorithm and explanation is provided; At step 10, the similarity between the two graphs is calculated by taking all paths similarity and dividing that by the number of all paths.

Algorithm 1 Vertex(node) and edge similarity for graph matching *sim(H,G)*

```
 1: procedure COVER MATCH(G, H)
 2:     while read line in G do
 3:         if G exists in Hn then
 4:             matchCount+1
 5:         end if
 6:         if G exists in He then
 7:             matchCount+2
 8:         end if
 9:     end while
10:     matchRatio = matchCount / totalLines
11:     return matchRatio
12: end procedure
```

Dataset used for matching

In this thesis a matching is used to classify malware into families, this matching is based on analysis of some of the malware in the dataset. When this methodology is used is it very important that the analyzed samples are excluded from the matching. To account for this the dataset was divided into a training set and a test set, this to separate the samples clearly. The background for this is that if analyzed train samples are used in the matching is will most probably be matched against itself and therefore present bias results. To manage this partition of samples, 4 random samples from each family was chosen to represent the training dataset, and the rest of the samples was used in the test dataset.

A common way testing in machine learning is to use cross validation. This is a method where a partition of a dataset is picked out, e.g., every 10th sample, to train the machine learning algorithm. Then when the testing of how well the algorithm has learned is tested on the whole dataset, including the training dataset. This method is not applicable in this thesis because of the bias occurring when dissecting and analyzing the test dataset therefore cross validation is not used in this thesis when conducting the matching.

7 Experiment

This chapter presents the experiment design and the reliability of the method and the experiment conducted. Finally, the experiment results are presented and evaluated for the different function layers.

7.1 Experimental design

This section presents the design of the of the experiment.

7.1.1 Reliability of the methodology

When dealing with a dataset that is not explored earlier, testing reliability of the dataset regarding false positives could be difficult. The reliability of the methodology represents how trusted the implementation of the methodology is. This was addressed in Section 3.2.1 where synthesize testing of obfuscation resilience and graph attributes extraction was described. Testing related to function call information based dependency matching was to some extent performed in [72, 73, 74], but more thoroughly tested by Sand in [9].

Since this has been tested thoroughly in a former thesis, this will not be conducted in this thesis. The results from Sand [9] presented in Section 3.2.1 will be used to derive the call graphs

These results are very applicable to this thesis because obfuscation and packing is a quite normal routine for malware to use, to hide main functionality. Moreover, understanding the effect of potential obfuscation is important. Sand [9] found the the methodology tested successfully against obfuscation and packing, therefore when the same methodology is implemented, this challenge of obfuscation and packing is not troubling.

7.2 Experiment setup and results

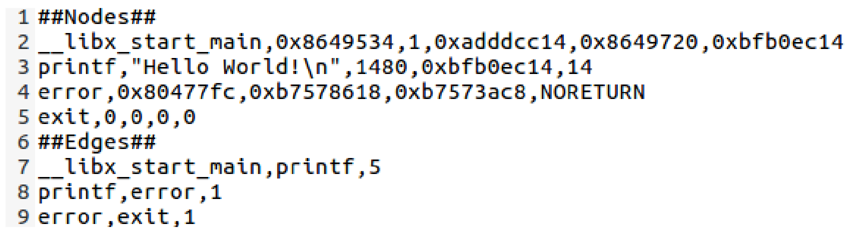
This section describes how the expert graph used in the matching is derived, this is done to make the structure of the graph files transparent and the matching understandable.

7.2.1 Preprocessing

To be able to utilize the acquired dataset, preprocessing of the dataset has been conducted to make the data readable and matchable. This was introduced in chapter 6, to get the dataset visualized and ready for matching, but to get an insight in the experiment further description of will be provided. The processing was mainly conducted to convert Ltrace and Strace output on a graph readable format and to present the traces in a matchable format.

Matchable format

To generate the matchable format the program in Appendix A is used to produce a comma-separated file (CSV). The input format, as described in Figure 22, is then rendered to a file containing a line for each node with the accompanying arguments <FunctionName,Arg1,Arg2,Arg3,ReturnValue>, and a line for each edge which the format <FromNode,ToNode,RelationValue>. This can be seen in Figure 25, where line 3, "Printf" is the the function name, "Hello World!
n" is the first argument, "1480" is the second argument, the hex value "0xbfb0ec14" is the third argument and finally, 14 is the return argument.



```
1 ##Nodes##
2 __libx_start_main,0x8649534,1,0xadddcc14,0x8649720,0xbfb0ec14
3 printf,"Hello World!\n",1480,0xbfb0ec14,14
4 error,0x80477fc,0xb7578618,0xb7573ac8,NORETURN
5 exit,0,0,0,0
6 ##Edges##
7 __libx_start_main,printf,5
8 printf,error,1
9 error,exit,1
```

Figure 25: Matching format after preprocessed

Figure 25 is the same program as presented in section 6.2.6 to make it easy to read. In line 2-5 the nodes of the trace are presented, and in line 7-9 the edges of the trace are presented. For the example to be more describing an alteration has been made to the memory addresses(ReturnValue) for the node in line 1. This has been done to exemplify one of the rules implemented in the matching format. As mentioned in section 3.2.1, if a return value is used as an input to an oncoming function call (node), there exist a dependency relation between the nodes. This has been marked on the edge between the nodes in line 7 with a relation type of "5", this will then be an input to the matching algorithm to regard when calculating the match ratio.

Comments to the matching format

When converting the graph notation to a matchable format some of the attributes are normalized, because of this it is important to point out what information which is preserved for the matching.

Firstly, it could be argued that the matching format is quite similar to what static analysis can extract from a malware sample regarding function calls. The difference regarding this must not be misunderstood. A static extraction of function calls will not gather information about arguments or return values to the calls, and the sequence of calls will not be counted for. Static analysis will reveal all functions in the code. Dynamic analysis will reveal which functions are used, the order of which they are used, the input arguments and the return value. Moreover, for packing or obfuscation of malware static analysis will not be able to extract all calls conducted by the malware, in contrast to how a dynamic analysis will.

Secondly, the match format describes a node with just the name of the function call. It can be argued that since, the call name is implemented in the edge and since all nodes will be included in the edges, nodes are superfluous. The matching includes both nodes and edges because a node match tells if the node regardless of relation to other nodes have been seen in the analyzed sample, and the edge tells if two nodes have been seen in relation to each other. There is a difference here which points out that an edge match is more describing for the sample analyzed. Although a node match is also describing for the sample in the context of function call overlap within a family, even if there is not an overlap in edges.

Thirdly, it is important to note that the implementation of the matching algorithm in Section 6.3.1 which uses the matching format described in Figure 25 does not fully implement detection in the ordering of calls. The algorithm does search for a simple node or edge matches, e.g., if a malware family has a common order of function calls ABC the algorithm will match for A, B, C individually, not all three in seen in the same context. It can be argued that the matching format has ordering dependency matching partly implemented, this because each edge is a relation between two nodes. This implies that the two nodes in an edge are related. This is a weakness of the matching witch would be very interesting to investigate in further work.

Expert graph deriving

To present the expert graph deriving, sections of how Neo4j, Gephi and a uniqueness matching will be presented.

7.2.2 Neo4j

Neo4j was the graph notation language used in this thesis because of the capabilities of visualization and graph querying. Neo4j is a graph database which can create, read, update and delete data in a graph, this is done by storing the information of the graph in a database where the relation between data points takes priority when storing the data. This is very convenient when working with a dataset based on graphs, and is used in this thesis to explore the data by querying for features and relations in specific samples. After a graph has been loaded into Neo4j, it is as mentioned possible to query the graph in different ways, this is done by the query language "Cypher" [111]. The language is SQL-inspired and allows to select and set requirements for what part of the graph to search for.

This has been utilized to look for specific patterns within the different families. For instance specific behavior of malware with a call within a range to another function call. This has been used to look for differences in families. To elaborate this, an example of the Cornelgen and Madvise family are presented.

In this example both samples has Internet connectivity functionality implemented, but they set up sockets and send data quite differently. So firstly a Cipher query can be used to look for connectivity related calls eg. a Send() function. The samples used is:

- 02eb5393ce1a84b800c5f5d26833e02f.cornelgen
- 3f00510ca077f4b821c84d9a3d5a3a41.madvise

Querying Send() nodes

```
MATCH (n:‘train/3f00510ca077f4b821c84d9a3d5a3a41.madvise.ltrace‘) where
    (n.callname)=’send’ RETURN n LIMIT 25
```

The result of this can be viewed in Figure 26.

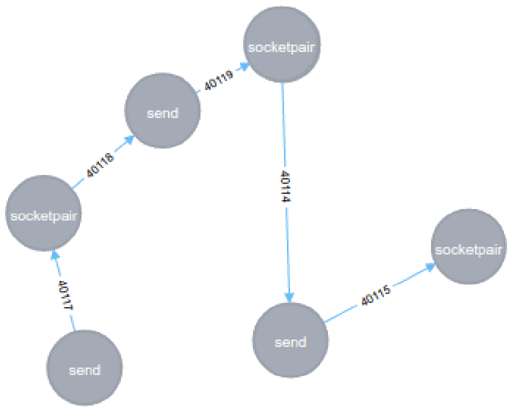


Figure 26: Results of Cypher query searching for Send() in Madvise sample

The same query for the other sample returns a similar, but a quite different results regarding the call names.

```
MATCH (n:‘train/02eb5393ce1a84b800c5f5d26833e02f.cornelgen.ltrace‘)
    where (n.callname)=’send’ RETURN n LIMIT 25
```

The artifact of importance in Figure 26 and Figure 27 is that the Madvise sample uses the function call "socketpair" to set up sockets while the Cornelgen sample uses "socket" to set up sockets. This can then be used to difference between the families if is a recurrent artifact about the family.

This is how Neo4j has been used to search for different artifacts for samples in families, and is used to build the expert graph describing each family. The main artifact that Neo4j has been used to extract from the training samples are Internet, registry, memory and string related calls.

- **Internet**: These calls was presented in the example in Figure 26 and refcorenelgenSend and will not be further discusses, except for an uncomplimentary list of calls that were investigated "inet_aton, inet_ntoa, inet_pton,

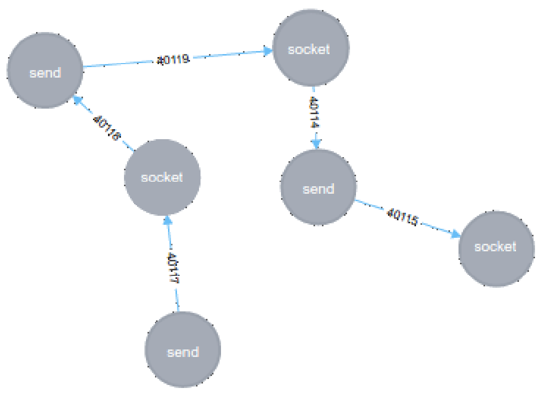


Figure 27: Results of Cypher query searching for Send() in Cornelgen sample

send, socket, socketpair, inet_addr, sock_host, gethostbyaddr, getnetbyaddr, getsockname".

- **Registry**: Malware often uses registry hives with accompanying keys to store configuration values, store data or achieve persistence. Unfortunately, the Linux operating system does not utilize registries, this is a Windows operating system specific implementation. Linux mainly uses configuration files along with the executable in the form of a plain text file. The way such files are read in Linux is by opening a file and read the content, this was too generic for all the samples to get any description of the families to include.
- **Memory**: Memory operations can be used in different ways in Linux. Some of the calls investigated were "malloc, realloc, memalign, memcmp, free". These are calls for using the memory allocated for the program running. The experiment showed that the Madvise family was the only to utilize the system call "memcmp" which is used to compare memory areas. This ended out in a call to be implemented in the expert graph.
- **Strings**: Strings can be handled by different functions in Linux. There are several calls to handle strings. When investigating strings, a clear difference was found in the different families. Linux has three methods for converting a string to a long integer; "strtol, strtoll, strtoq". Among these three "strtol" was only seen in the Madvise family and "strtoq" was only seen in the Rst family, this was therefore implemented in each of the expert graphs amongst other string calls.

Another functionality implemented in Neo4j is the shortestpath() function which can be used to find the least amount of links between two nodes. Using the shortestpath function so search for paths between two specific nodes can point out how a family utilize different nodes (function calls) related to each other. This was not conducted, because the graph description in this thesis does not take ordering dependency into account in such manner when building the graph. Searching for such dependencies should be considered for further work.

7.2.3 Gephi

Gephi is an open graph visualization platform and has its main focus at presenting graphs and to explore graph data, including nodes and edges. Gephi has the functionality to analyze sections of a graph and divide it into networks or sub graphs with similar features and statistical functionality to do calculation such as betweenness, centrality, closeness and clustering of graphs [112].

In this thesis, gephi was used to look for internal structure in each family, and in total. It is a possibility that statical functionality could reveal this by applying implemented graph analysis such as clustering for the whole training dataset, this was tested and will be described to present the possibilities of gephi regarding graph analysis.

Since the dataset already was converted to a Neo4j readable format, gephi was able to read a plain export of this data. All of the data from the training samples was exported into gephi with a label and color for each node to separate each family. In Figure 28 the size of each node represents how many times the call is used (call name not shown in the figure) and the thickness of the link represents how many times the link between the relevant nodes is seen.

As seen in Figure 28 there is no obvious linking between the different families, but there are some features in the graph that is worth noticing. Firstly there is an overweight of purple nodes, this is nodes linked to the Tsunami family. From this it can be assumed that Tsunami, in general, has more calls for each sample that is traced, which is consistent with the dataset statistics table 3. This can then be used to investigate the strongly connected nodes further, by looking at the call name for the bigger nodes and the see if they are connected to the same family or if it is a common call for all families. An example of this is presented in Figure 29.

In Figure 29 the connected nodes to a specific node is highlighted to see visualize the connectivity to other nodes. As seen in Figure 29 there is a majority of connected nodes of the same family (Tsunami), because of this relationship the node can be used as a call inserted in the expert graph to look for this family.

A second characteristic with the full node Figure 28 is the prominent link between two nodes reaching from the down right corner up to the middle left. This is the line with arrows at both sides. When a relation is seen often a thicker line is drawn, to explore this, the call names and relation is investigated. As shown in Figure 30 this relation is a inter-family, connecting a call related to the Madvise and Cornelgen family.

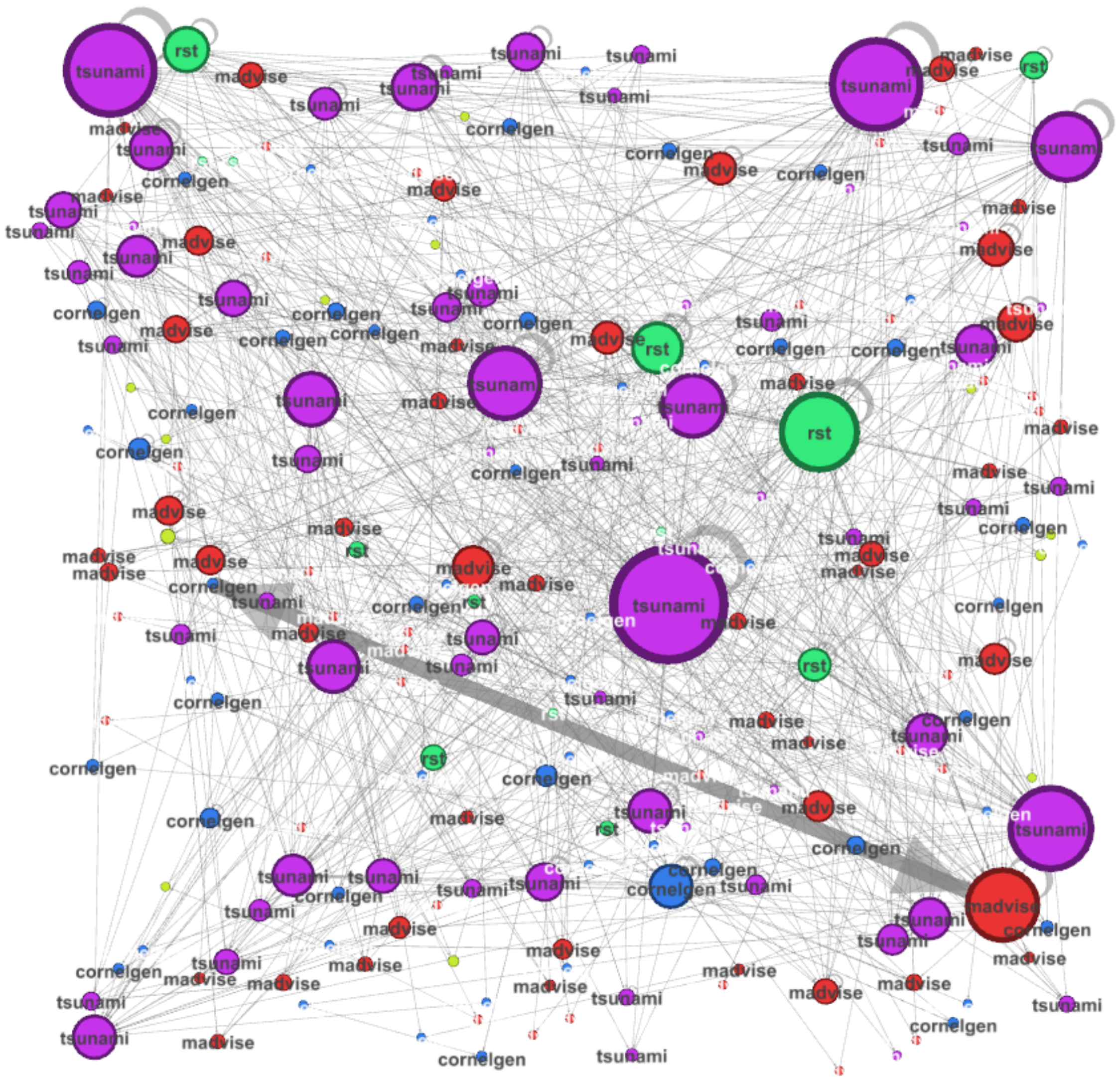


Figure 28: Visualization with Gephi of all train samples

In Figure 30 the relation is marked in red color. When focusing on this node it also shows that the node is related to a lot of other calls in all of the families, this indicated that this might not be a good selector to distinguish between families. The call names for the strongly connected nodes is "strcmp" and "fscan", which is two standard functionality function calls for many programs comparing and reading strings. This points out that this will not be a good indicator for family classification, since it is so widely used also within the relevant malware families.

A third characteristic with the full node Figure 28 is that some of the nodes

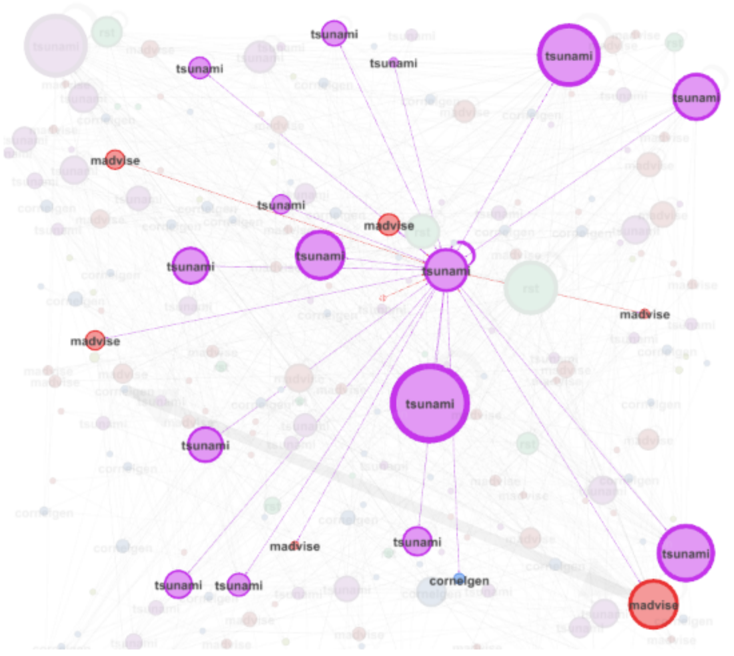


Figure 29: Visualization with Gephi focusing on a specific node and relations

have a strong link to it self. This means that the same call has been used several times in a row or are connected strongly to itself. This is shown in the Figure 31 with a self returning edge.

For this node the call name is "setlocale" which is used to query the programs current local path, this is a commonly used function for programs, this indicates that this may not be a suitable call for the expert graphs. On the other hand, if the other families do not utilize this call to do this, or uses other similar functions to get the local path it could be used. To overcome this overlapping problem, function call matching for the different families are described later in this thesis, in Section 7.2.4.

Gephi clustering test

Gephi has graph analysis functionality, including clustering algorithms. One of the included clustering algorithms is Markov clustering, which is a fast, scalable cluster algorithm for graphs. The result of this unsupervised clustering was 23 clusters with 1-2 nodes in each cluster, and there were no families that were regulars for each cluster.

Since this gave imprecise results and there was no easy way to test this further with the experiment setup for this thesis and because this is not the main focus of the thesis this will not be further explored, even though this is a very interesting area of further work.

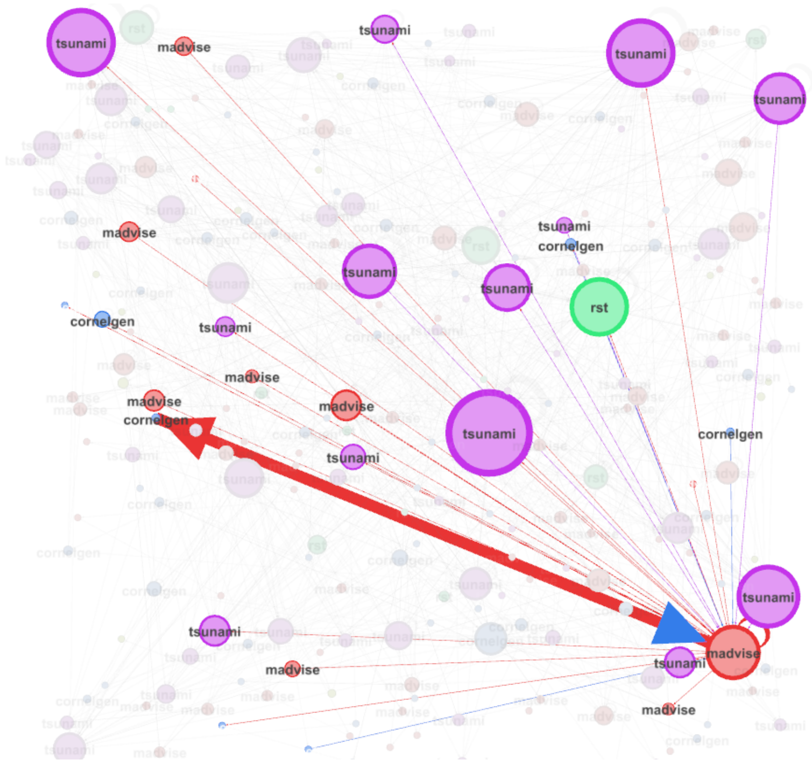


Figure 30: Visualization with Gephi focusing on a specific relation

Figure 31: Visualization with Gephi focusing on a self returning edge for node

7.2.4 Graph uniqueness matching

Graph uniqueness matching is, in this case, a matching of all the graph attributes in the training set to look for uniqueness within a family. Graph uniqueness matching was addressed to overcome the challenge of overlapping nodes between the families. This was conducted because when looking at Figure 32 which presents node names (call names) and the link between nodes, there are many calls that at first sight looks like a specific node for a family. Figure 32 presents the training data with call name as the label and the size a node represent how much it is used. The color of the node still represents the family it is related to.

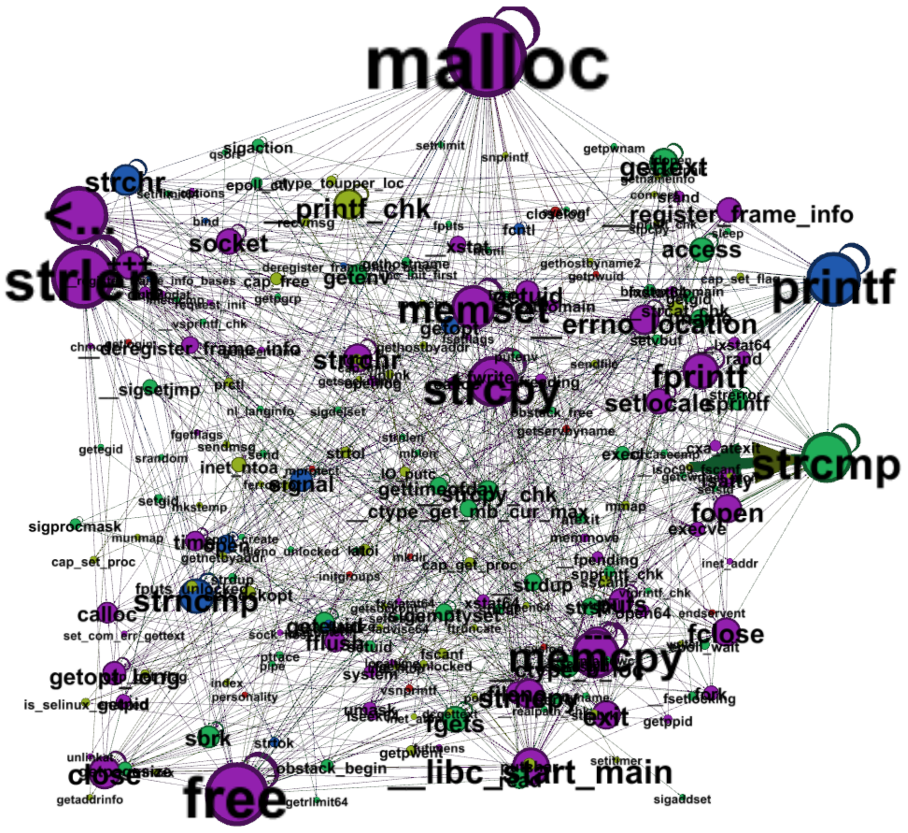


Figure 32: Visualization with Gephi with call names as label

As seen in Figure 32 the node with the label "malloc" at the top of the graph stands out, it has a big label and is a big node. This means that this call is seen several times in the family Tsunami (purple). Other nodes that stands out is "printf, strcmp, strlen, memcpy, free" which all is quite normal function calls in many programs. This addresses the problem mentioned in section 7.2.3 where some calls looked specific to a family, but since the calls are standard functionality in programs they could either be confirmed to function as a usable entity or not.

To conduct the graph uniqueness matching all nodes and edges for each family was listed, this list was then altered to only contain unique nodes and edges. This was done for all five families before they where matched against each other, by utilizing the Linux binary "comm". By listing all unique entities for a family, the unique calls could be used to populate the expert graph.

7.2.5 Matching program output

To give insight in the graph matching which is implemented by the program in Appendix B, exhaustive output from each step of the matching is shown in an example. A test sample is run in the matching program, and the output is presented and explained.

```
1.  python match.py
    test/8e916c244664eef8d60b93afeb581a8e.tsunami.ltrace.csv
2.
3.  +++ Match nodeSimple +++
4.  Match Counter: 0.0 Match ratio: 0.0         Aself
5.  Match Counter: 10.0 Match ratio: 0.040160   Cornelgen
6.  Match Counter: 0.0 Match ratio: 0.0         Madvise
7.  Match Counter: 0.0 Match ratio: 0.0         Rst
8.  Match Counter: 16.0 Match ratio: 0.064257   Tsunami
9.  +++ Match edgeFull +++
10. Match Counter: 0.0 Match ratio: 0.0         Aself
11. Match Counter: 0.0 Match ratio: 0.0         Cornelgen
12. Match Counter: 0.0 Match ratio: 0.0         Madvise
13. Match Counter: 0.0 Match ratio: 0.0         Rst
14. Match Counter: 5.0 Match ratio: 0.020080    Tsunami
16. Nodes: [0, 10, 0, 0, 16]
17. Edges: [0, 0, 0, 0, 5]
18. Class: [5, 5, 5]
20. Tsunami Classifed as: Tsunami
21. CORRECT!!
```

When a test sample is matched against the expert graphs, this is done in two steps. Firstly the nodes are matched then the edges are matched. The matching is implemented this way to separate the types and keep an overview of which type that has most matches. The "match counter" is a metric accumulated for each hit the test sample has in the different expert graphs. For each line in the test sample, a counter is started to keep hold of the total entities of nodes in the test sample, this is used to calculate the match ratio (matchCount/totalEntities). This is shown in the listing in line 3-8, where the different families expert graph is added to the end of the line. In line 9-14 the exactly same matching is done for the edges of the test sample.

The result of the matching results in an array representing the five families <[aself, cornelgen, madvise, rst, tsunami]>. This array contains the count for each family, this is presented in line 16 and 17. Subsequently, a "Class" array is created, this array is where the weighting is included, this array consists of the results of the edge end node matching and is built as <[node, edge, edge]>. By doing this, the edges is weighted more then a node match as presented in Section 6.3.1. In this array each family has its own value; Aself:1, Cornelgen:2, Madvise:3, Rst:4 and Tsunami:5. In the example above both nodes and edges is classified as Tsunami therefore the arrays in line 18. is presented as "[5,5,5]".

This array is then tested to see if it match the correct affiliation of the test sample, if its correct the "CORRECT!!" string printed.

The reason weighting is implemented in this thesis is because of an edge is more descriptive than a node since it describes the relation between two nodes. How the weighting is implemented is not trivial and is not thoroughly tested, this because of time resource limitation within the frame of a master thesis. Weighting in a matching algorithm can be very hard to implement, and to optimize the classification accuracy for matching. Because of the apparently obvious difference between nodes and edges, we have chosen to implement different weighting. The actual weighting used should be considered more an acknowledgment of the importance of the difference rather than an attempt to get weighting correctly implemented. Optimizing the weighting is future work.

7.3 Experimental classification results

The purpose of this section is to cover the results from the experiments. In other words, present the classification results for the multinomial classification of malware based on call graph matching. The difference for system calls, library calls and a pairing of both results called hybrid calls will be discussed, and the total classification rate will be discussed.

7.3.1 System calls

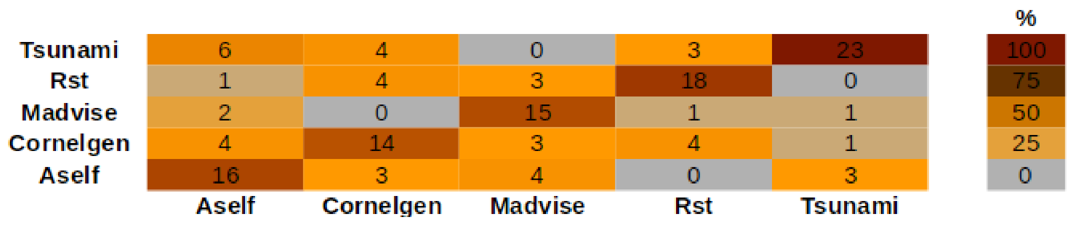
The classification accuracy for system calls was in total 69,92 %. This it the result of a multinomial classification with 5 outcomes. For system calls there where 40 faulty classified samples out of a total 133, these wrongly classified samples was spread out between the different families. Table 4 is used to summarize the results in total for this layer of classification.

Table 4: System layer classification accuracy

System	**Classified as**	
	Positive	Negative
Correct class	93	40
Classification accuracy	**69,92 %**	

The classification accuracy is calculated by taking all correctly classified samples divided by the total amount of samples. Table 4 presents the big picture for the classification at the system layer, but since this is a multinomial classification, it is interesting to see if the different malware families are classified into. To present this, a heat map between the malware families and classification is used to visualize this properly.

The heat map in Figure 33 presents the classification with a color dependency. This is calculated with a percentage hit of correctly classified samples to the correct family. The number in each cell presents the number of samples placed in the specific place in the heat map matrix. The matrix is based on an X-axsis



	Aself	Cornelgen	Madvise	Rst	Tsunami
Tsunami	6	4	0	3	23
Rst	1	4	3	18	0
Madvise	2	0	15	1	1
Cornelgen	4	14	3	4	1
Aself	16	3	4	0	3

%
100
75
50
25
0

Figure 33: Presentation of Strace results in a heat map

and a Y-axsis representing each family, if Aself sample is classified as Tsnuami the matrix will increment the number in cell <Aself, Tsunami> by one. Moreover, the darker the color, the higher the related value. At the right in Figure 33 the color scheme for percentage hit is shown. Heat maps will be used for all layers with the same functionality.

The classification matrix for system calls have a clear correlation for each family where the correct classification marked, form bottom left to top right.

All over this is a quite satisfying results whit the most data point at the right family entity with some outliners. The Tsunami family stands out as the family with the most correct classifications. Also at the top left corner, there have been some Aself samples classified as Tsunami samples. This indicated that the expert graphs for these have some of the same entries for detection, which nodes and edges that created this overlap is very interesting. An analysis of this is not implemented in the matching program used in this thesis, but further analysis of this would be very interesting.

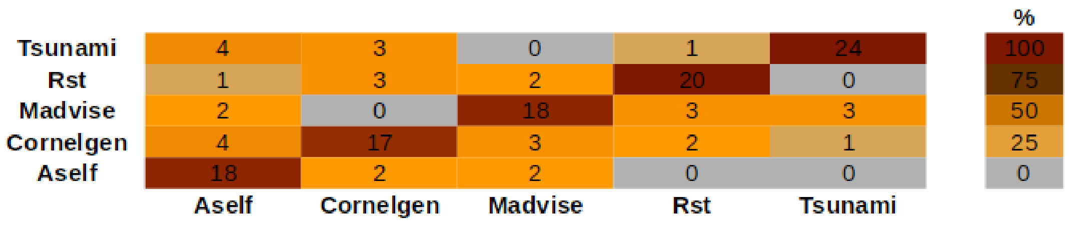
7.3.2 Library calls

The classification accuracy for library calls was in total 77,44 %. The classification rate is 7,5 % better than for system calls. For library calls there where 30 faulty classified samples out of a total 133, these wrongly classified samples was spread out between the different families. Table 5 and Figure 34 is used to describe the results further for this layer of classification.

Table 5: Library layer classification accuracy

Library	**Classified as**	
	Possitive	Negative
Correct class	103	30
Classification accuary	77,44 %	

The classification accuracy is calculated in the same manner at for system calls. Table 5 presents the big picture for the classification at the system layer, and a heat map of the classification is also presented for the library layer to present the distribution between families.



	Aself	Cornelgen	Madvise	Rst	Tsunami
Tsunami	4	3	0	1	24
Rst	1	3	2	20	0
Madvise	2	0	18	3	3
Cornelgen	4	17	3	2	1
Aself	18	2	2	0	0

%
100
75
50
25
0

Figure 34: Presentation of Ltrace results in a heat map

For the library level, the results are even more successful than for the system layer. This is visualized with the line from the lower left corner to the top right corner. The heat maps are quite alike, but the main difference is that the library layer has fewer falsely classified samples and therefore a stronger color distribution for the correct classification. The overlap between Tsunami samples and Aself samples is not present in the same manner as for the system classification, this points out that the expert graph is more distinct for the library level than for system level.

7.3.3 Hybrid calls

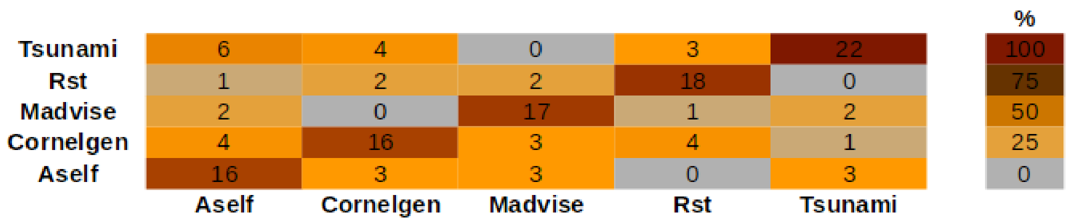
A third classification layer can be accumulated in this experiment, the layer is named hybrid calls and is a crossing of the system an library layer results. This result was calculated by aggregating the node hits for system and library level and then for the edges as well. Referring to the matching output presented in Section 7.2.5 the results for each layer was counted in an array for the nodes and the edges. The sum of these arrays is the data used to generate the hybrid class. To classify the hybrid class, these new arrays are matched against the same expert graphs as the system and library layer.

The classification accuracy for hybrid calls was in total 72,18 %. This is 5,26 % less than for library calls and 2,2 % higher than for system calls. Table 6 and Figure 35 is used to describe the results further for this layer of classification.

Table 6: Hybrid layer classification accuracy

Library	**Classified as**	
	Possitive	Negative
Correct class	96	37
Classification accuary	**72,18 %**	

The classification accuracy is calculated in the same manner at for the other layers. Table 6 presents the big picture for the classification at the system layer, and a heat map of the classification is also presented for the library layer to present the distribution between familes.



Tsunami	6	4	0	3	22
Rst	1	2	2	18	0
Madvise	2	0	17	1	2
Cornelgen	4	16	3	4	1
Aself	16	3	3	0	3
	Aself	Cornelgen	Madvise	Rst	Tsunami

%
100
75
50
25
0

Figure 35: Presentation of hybrid calls results in a heat map

7.3.4 Summary

The classification results for each layer it presented in previous sections, to summarize this, the best classification rate was achieved using library calls with a classification rate of 77,44 %. The comparison of the three classes can be viewed in Table 7, where each layer with the accompanying classification accuracy is listed. This result is quite satisfying when taking into account that a classification has five possible outcomes.

The hybrid calls have an accuracy in between the system and library level, this is to some extent surprising, because by merging the results, more attributes are available to classify each sample and could enhance the accuracy. However, since the hybrid layer is a summarization of both results, the accuracy is closer to system calls then to library calls. This indicates that it is not a synergy between the two layers, on the other hand, it lower than the mean classification accuracy of both classes. This could be because the system layer has more nodes and edges which will overweight in a plain aggregation of hits. In Table 3 the statistics to reinforce this theory is presented. The impact of the summarization will be discussed in the Discussion Chapter 8.

The results strengthen the theory presented in Section 3 where it was presented how library calls utilized underlying system calls when they execute, which will result in less variety in system calls than in library calls.

Table 7: Summary of classification accuracy

	Classification accuracy
System layer	69,92 %
Library layer	77,44 %
Hybrid layer	72,18 %

8 Implications and discussions

This chapter provides a discussion of what implications the results of the experiment has, and presents different aspects of the work conducted.

Dataset
For this thesis, a new dataset was generated by downloading malware for the site VirusShare [94]. This site is a well established malware collecting site but is not commonly used for research. By creating a new dataset the opportunity to compare results to other work is difficult, this because the nature of datasets is different. For the results to be compared to other experiments, the dataset must be evaluated to see if they are actually comparable, or the same dataset must be used. Another important aspect of generating a new dataset is that future work is not directly comparable if the same dataset it not used. As an example L.A. Sand [9] conducted malware classification based on information based dependency matching, Sands results can not directly be compared to the results of this thesis because Sands dataset did not divide malware into families.

Graph extraction
This thesis investigates the possibility to classify malware into families by building call graphs for all analyzed samples. Extraction of the call graph was conducted by tracing executed malware and record all function calls at different layers. Tracing executed programs is not always trivial, and in this thesis, tracing is used to gather function calls to get a fingerprint of an malware. Both Windows PE32 files and Linux ELF files were used to generate call graphs in this thesis, PE32 was initially the malware file type tested. As presented in Section 6.2.6 tracing PE32 files to generate call graphs generated challenges with the applied method.

The main problems with PE32 files were to get the malware samples to execute properly. Section 6.2.6 describes environmental problems and understandability of tracing output. The tracing problems described in Section 6.2.6 is related to call graph extraction using IDA pro with additional plugins and scripts. An interesting observation is that there is a lot of sandboxing products and applications that conduct automated dynamic analysis of malware, and can extract function calls flawlessly. In this thesis the problems were related to how IDA pro attaches to a malware sample, another approach could be to use well established malware analysis sandboxes which have call graph functionality implemented.

Graph Matching

Graph matching theory was described in Chapter 5 and the chosen graph matching approach was presented in Section 6.3.1. The theory noted that the best approach for graph matching is an inexact graph matching, this was used by implementing a Node and Edge cover algorithm approximation. A comparison of different graph matching approaches is not conducted in this thesis because previous work has pointed out a preferable approach, which is used in this thesis, but it would be interesting to investigate the difference between multiple approaches to reveal practical implications.

The node and edge cover algorithm described in Section 6.3.1 is quite simplistic. Shortly described it adds each match for nodes and edges and divides the total amount of matches on the total nodes and edges, included a weighting for nodes. The reason weighting is implemented in this thesis is because an edge is more descriptive then a node, this because the edge describes the relation between two nodes. Finding the best performance for weighting is very hard, the weighting in this thesis is simplistic and is not thoroughly tested, this because of time resource limitation within the frame of a master thesis. Weighting in a matching algorithm can be very hard to implement to optimize the classification accuracy for matching. Testing different values for the weighting to see how this affects the classification accuracy is definitively a domain that should be addressed thoroughly, but because of the obvious difference a node and an edge match a simple implementation considered as a usable option.

Expert graph derivation

Expert graphs were built for families based on a training set for each malware family in this thesis, the derivation of these graphs was based on a manual analysis. Neo4j, Gephi and a uniqueness match were used to build these graphs. These methods are described in Section 7.1 and are based on three different analysis approaches. Neo4j was used to search for expected dependencies and differences in calls and relations between families based on common knowledge describe the theory of Chapter 2 of malware behavior. One important aspect of this approach is that this methodology will only provide findings related to knowledge acquired in advance. This reflects the expert knowledge and theoretical research used in this thesis. This approach was supported by the next analysis approach using Gephi, by visualizing the dataset Gephi facilitates for exploration of the dataset.

The utilization of Gephi can be described as superficial to some extent, because there are approximately unlimited ways to visualize large datasets. The methodology was to visualize the different families and search for protruding artifacts of the representation. This approach depends on the visualization being correct and not being a random layout, since this would create a different analysis environment for each time the dataset is loaded.

The last methodology used was to exclude nodes and edges in the expert graphs which appeared in more then one expert graph. This was used to exclude overlapping description of the families. It can be argued that this is not necessary for increasing the classification rate, since even though a node is present in multiple expert graphs, it may be more describing for one family than another. This can result in a wrongly classification of a sample, but it may also help to distinguish relevant family from the other families. An example would be if a node is present in both expert graph A and B, and an incoming sample matches for both these graphs, but it since families C,D and E does not contain the node three families are excluded. Subsequently, the incoming sample has more matches for both family B and C, this will then result in that the sample is classified as family B and not A or C, even though the node is present in both A and B.

Classification

The classification in this thesis is multinomial, with five outcomes. The classification accuracy was presented for the three layers system, library and hybrid in Section 7.3. The classification accuracy for these layers where respectively 69,92 %, 77,44 % and 72,18 %. These results point out that the library level has the highest classification accuracy, a reason for this is related to the theory presented in Chapter 5 where it was presented how library calls utilized underlying system calls when they execute, which will result in less variety in system calls then in library calls.

For the classification accuracy for the hybrid calls, the results are in between the system and library layer. This is to some extent surprising since this is a summarization of both results, but the accuracy is closer to system calls then to library calls. This indicates that it is not a synergy between the two layers, in the way the matches are accumulation in the matching methodology. On the other hand, it lower than the mean classification accuracy of both classes. This could be because the system layer has more nodes and edges which will overweight in a plain aggregation of matches. The accumulation of system and library matches to generate the hybrid layer matches needs to be addressed, since this is directly affecting the classification accuracy. This is future work.

Firstly, the method does not take into account that the ratio of matches within a family. This means that if an incoming sample is compared against two expert graphs A and B with respectively 20 and 10 matching entities, a match for 11 out of 20 entities in expert graph A will surpass 10 out of 10 matches in expert graph B. Even though the match percentage is higher in expert graph B. This can be avoided if each match count is calculated as a ratio of the total entities in the expert graph.

Secondly, the variety of matches in an incoming sample is not taken into account. An example for this is that if a sample has 10 matches with 10 different entities in expert graph A, it will be surpassed by a 11 matches for the same entity in expert graph B. This can be avoided by only counting each entity match once for each expert graph.

Thirdly, as presented in the dataset statistical description in Table 3, the system layer has more nodes and edges in general then the library layer. When the results of these matchings are simply added together to create the hybrid layer, contribution for system nodes and edges will be in overweight compared to library nodes and edges as mentioned. This will weight the system layer results more then the library level, which is undesirable considering the classification accuracy for the different layers. To cope with this impact, the matches could be normalized with a score for the system and library level based on some nodes and edges used.

The mentioned examples are very connected to the discussion relating graph matching in the start of this chapter, but is addressed in this paragraph because it clearly is shown by the results in the classification accuracy. These three methods may improve the classification accuracy and would be interesting to test, but because of time limitations of this thesis they where not implemented, and is left for future work.

9 Conslusion

This thesis has built upon the existing idea of using function call graphs to describe malware behavioral. Sand [9] built call graphs to classify samples as either malicious or benign, using machine based graph learning. This thesis addresses the same domain, but with another approach. The classification was not binary but multinomial and classified malware into different malware families, and the classification was based on expert knowledge. This means that a part of the dataset was analyzed and an expert graph was described for the different malware families. The incoming unclassified test samples, were then classified utilizing inexact graph matching, and not machine based graph learning.

Both Windows PE32 and Linux ELF files were initially tested in this thesis, with varying results. Call graph extraction of PE32 files using the debugger software IDA pro resulted in difficulties related to execution and tracing of samples, the occurring challenges was related to environmental setup, successful execution of samples and complexity and overhead in output. Some of the same challenges occurred related to ELF samples, but not to the same extent, therefor ELF samples were used as the file type for malware classification.

The experiment conducted were based on a newly generated dataset of approximately 150 samples, which was divided into five families. For the total dataset 3-5 samples from each family was randomly chosen to be a part of the testing dataset. These samples was analyzed and used to derive the expert graphs for each family. The derivation of expert graphs was based on three methods that successfully pointed out differences between the families. Firstly, Neo4j was used to search for how the families handled Internet connections, strings, memory and registry operations. Secondly, Gephi was used to visualize the families represented as graphs and exploring protruding artifacts. Thirdly, a comparison of the expert graphs with each other to ensure uniqueness for the different families.

The experimental results presented a classification accuracy of three different layers system, library and hybrid. The accuracy for these layers varied some, but was in general good for a five-fold multinomial classification, with result varying for 69 - 77 %. The best classification accuracy was at the library layer and the weakest at he system layer. A random classification of samples into five families would, in general, give a correct percentage of 20 % due to the nature of randomly guessing the class of a sample. This must be considered when comparing the results to other classification accuracy, e.g., if there is a binary classification between benign and malicious samples where the random guess rate is 50 %.

Throughout the experiment, a quite simplistic matching algorithm based on inexact graph matching was used. When analyzing the results, assumptions about weaknesses related to the implementation of this algorithm was discovered and addressed. This points out drawbacks of the matching which could enhance the classification accuracy, and can be seen as a contribution for considering what matching methodology to consider in future work.

In conclusion, this thesis has shown a methodology for malware family classification using call graphs. It has presented successful and unsuccessful extraction of call graphs and described related problems to each method. By doing this, in total, this thesis has made progress in testing which methodology is preferable when conduction malware classification based on function call graphs.

10 Further work

Malware analysis is not an entirely new research domain, and there are several directions of research related to the analysis of function calls extracted from dynamic analysis. The most common targeting is to distinguish between malware and benign software. However, not many researchers focus on differentiation between malware families using graph matching. Therefore this is a research domain which requires further research to explore the domain. In this final chapter problems and issues stumbled upon during this thesis will be discussed to clarify further work for the research.

First of all the work conducted in this thesis regarding Windows PE32 files should definitively be continued. The experiments conducted in this thesis failed during the execution of PE32 malware samples with IDA pro. By utilizing other analysis software, this may be more successful. Using well established sandbox appliances which can trace function call and extract the information required to build call graphs, should be considered. The focus on PE32 files is very important because of the world wide spread of the file type. Windows malware is the most common type of malware because of its vast user mass. The dataset of PE32 files in this thesis was a subset of the one used by Shalaginov et.al [104], this is a big dataset of malware samples predefined into families. Interesting work related to this dataset, and this thesis, is to search for environmental dependencies within this dataset. This can reveal what is needed to create an environment suitable for executing malware within the different families.

Another topic for further work is to experiment with different graph matching approaches and techniques. Firstly, work related to testing of exact and inexact graph matching would be interesting. If this is done utilizing the same dataset, this can reveal differences in classification accuracy and can provide a clarification of how big the difference is. Secondly, further work with a matching algorithm can be performed to increase the classification accuracy. This includes implementing orderly matching. This means take the order of nodes and edges into account when matching against an expert graph. For example, if an incoming sample contains entities A,B,C connected to each other, and expert graph A contains A,B,C and expert graph B contains A,C,B. The order of the entities can help to classify this sample as type A, since the order of entities is more describing. Thirdly, further work regarding the weighting between nodes and edges used in the graph matching algorithm can be tested to optimize classification accuracy. A correct weighting is often very hard to implement, testing for optimal weighting parameters is comprehensive work and where not addressed in this thesis. The last notice for further work related to the graph matching is to utilize matching tools or frameworks to conduct the matching, tools as GraphLab [113] or Neo4j [14] can be used to implement matching. This requires deep knowledge of these applications work to be able to utilize and configure the matching correctly.

Further work related to how the expert graphs are derived should be considered. The methods used to derive the expert graphs in this thesis are quit superficial, the derivation is based on analysis of a test set of samples. This result of this analysis is dependent on the knowledge and creativity of the analyst, therefore thorough analysis searching for interdependencies in the malware families is person based, and should therefore be conducted by various people with a variety of skills. Another aspect related to the result of this analysis, is how the expert graphs are distinguished from each other in this thesis. After analyzing all families and creating expert graphs, the expert graphs are compared to each other searching for nodes and edges appearing in multiple families. These nodes are then removed from the family which utilized these nodes or edges the least. This challenge was addressed in Section 8 and further work to clarify how the classification accuracy is affected by implementing this would be interesting.

Lastly, this thesis used a dataset where malware was divided into families. Interesting further work would be to see if the same methodology can be used to in a multinomial classification of malware types.

Bibliography

[1] Lenny zeltser, mastering 4 stages of malware analysis, 2015. https://zeltser.com/mastering-4-stages-of-malware-analysis/. Accessed: 2016-08-21.

[2] Sectechno, automating malware analysis cycle 2011. http://www.sectechno.com/automating-malware-analysis-cycle/. Accessed: 2016-08-21.

[3] K. franke and l. sand, reverse engineering malware, 2015, slides from øyskolen i gjøvik imt4022 lecture 8. https://www.ntnu.no. Accessed: 2016-08-20.

[4] Michael Sikorski, A. H. 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. William Pollock, 1. edition, https://www.amazon.com/Practical-Malware-Analysis-Hands-Dissecting/dp/1593272901.

[5] Josh grunzweig, pe file structure, header and sections 2013. https://www.trustwave.com/Resources/SpiderLabs-Blog/Basic-Packers--Easy-As-Pie. Accessed: 2016-08-20.

[6] Microsoft, user mode and kernel mode, 2016. https://msdn.microsoft.com/en-us/library/windows/hardware/ff554836(v=vs.85).aspx. Accessed: 2016-05-11.

[7] man Pages, F. 2004. Manual reference pages - elf (5). http://www.gsp.com/cgi-bin/man.cgi?section=5&topic=elf.

[8] The Santa Cruz Operation, I. 2007. Elf section description, 'sections'. http://refspecs.linuxbase.org/elf/gabi4+/ch4.sheader.html.

[9] Sand, L. A. 2012. Information-based dependency matching for behavioral malware analysis. https://brage.bibsys.no/xmlui//bitstream/handle/11250/143994/LASand.pdf?sequence=1.

[10] Doug swanson, the cat-and-mouse game: The story of malwarebytes chameleon, 2012. https://blog.malwarebytes.org/cybercrime/2012/04/the-cat-and-mouse-game-the-story-of-malwarebytes-chameleon/. Accessed: 2016-05-10.

[11] McAfee. August 2015. Mcafee labs threats report. http://www.mcafee.com/us/resources/reports/rp-quarterly-threats-aug-2015.pdf.

[12] Manuel Egele, T. S. February 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(6). http://dl.acm.org/citation.cfm?id=2089126.

[13] Kinable, J. 2010. Malware detection through call graphs. https://daim.idi.ntnu.no/masteroppgaver/005/5741/masteroppgave.pdf.

[14] Group, T. N. 2017. Neo4j: The world's leading graph database. https://neo4j.com/.

[15] Saumya K. Debray, Kevin P. Coogan, G. M. T. 2013. On the semantics of self-unpacking malware code. https://www.cs.arizona.edu/ debray/Publications/self-modifying-pgm-semantics.pdf.

[16] Sebastien andrivet, advobfuscator plugin, 2016. `https://github.com/andrivet/ADVobfuscator`. Accessed: 2016-05-10.

[17] Bishop, M. 2003. *Computer Security: Art and Science*. Addison Wesley Professional, 1. edition, https://www.amazon.com/Computer-Security-Art-Science-Set/dp/013428951X.

[18] The-1020Group. May 2015. Clustering analysis of malware behavior. http://projekter.aau.dk/projekter/files/213481767/Report_15gr1020.pdf.

[19] Wikipedia, brain virus desciption site, 2009. `https://en.wikipedia.org/wiki/Brain_(computer_virus`. Accessed: 2016-08-18.

[20] Ricky, M. June 2016. The return of macro attacks. http://www.windowsecurity.com/articles-tutorials/viruses_trojans_malware/return-macro-attacks.html.

[21] Wikipedia, logic bomb wiki site, 2012. `https://en.wikipedia.org/wiki/Logic_bomb`. Accessed: 2016-08-19.

[22] Zetter, K. March 2013. Logick bomb set off south korea cyberattack. https://www.wired.com/2013/03/logic-bomb-south-korea-attack/.

[23] Ravula, R. R. August 2011. Classification of malware using reverse engineering and data mining techniques. https://etd.ohiolink.edu/!etd.send_file?accession=akron1311042709 &disposition=inline.

[24] The free dictionary inc., sobig worm, 2012. http://encyclopedia2.thefreedictionary.com/Top+10+Worst+Computer+Worms+of+All+Time. Accessed: 2016-10-31.

[25] A. Kalafut, A. A. May 2006. A study of malware in peer-to-peer networks. *ACM*. http://dl.acm.org/citation.cfm?id=1177124.

[26] Dan Farmer, W. V. 2004. *Forensic Discovery, book abouth forensic methdodology*. Addison Wesley, 1. edition, http://www.fish2.com/security/wf-book.pdf.

[27] Wikipedia, user space wiki site, 2015. https://en.wikipedia.org/wiki/User_space. Accessed: 2016-08-20.

[28] Wikipedia, kernel (operating system) wiki site, 2014. https://en.wikipedia.org/wiki/Kernel_(operating_system. Accessed: 2016-08-20.

[29] Aycock, J. 2006. *Computer Viruses and Malware*, volume 22. Springer, 1. edition, http://www.springer.com/gp/book/9780387302362.

[30] My toolbar is infected, list of toolbar adware, 2014. https://infectedbrowser.wordpress.com/list-of-adware/. Accessed: 2016-10-31.

[31] Adaware, cryptowall ransomware cost, 2016. http://www.lavasoft.com/mylavasoft/company/blog/cryptowall-ransomware-cost-users-325-million-in-2015. Accessed: 2016-10-31.

[32] Berry, A. May 2017. Wannacry malware profile. https://www.fireeye.com/blog/threat-research/2017/05/wannacry-malware-profile.html.

[33] Sanchez Squillero Tonda Cani, G. 2013. *Towards automated malware creation: Code generation and code integration*, volume 22. 1. edition, http://www.cad.polito.it/downloads/White_papers/Towards

[34] Schiffman, M. 2010. A brief history of malware obfuscation: Part 1 of 2. http://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2.

[35] Joshua cannell, obfuscation: Malware's best friend, 2013. https://blog.malwarebytes.org/intelligence/2013/03/obfuscation-malwares-best-friend/. Accessed: 2016-08-20.

[36] Axelle, R. 2014. Obfuscation in android malware, and how to fight back. https://www.virusbtn.com/virusbulletin/archive/2014/07/vb201407-Android-obfuscation.

[37] Differencebetween website, difference between obfuscation and encryption, 2009. http://www.differencebetween.info/difference-between-obfuscation-and-encryption. Accessed: 2016-08-20.

[38] Mike, S. February 2010. A brief history of malware obfuscation: Part 1 of 2. https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2.

[39] Wanner, R. 2007. Malware analysis: Environment design and artitecture. https://www.sans.org/reading-room/whitepapers/threats/malware-analysis-environment-design-artitecture-1841.

[40] Walenstein, Mathus, C. L. 2007. The design space of metamorphic malware. http://vxheaven.org/lib/aal02.html.

[41] Schiffman, M. 2010. A brief history of malware obfuscation: Part 2 of 2. http://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_2_of_2.

[42] Media, S. 2008. Malware delivery – understanding multiple stage malware. http://security.sys-con.com/node/2483992/mobile.

[43] NSA. 2016. Ant product data. https://www.eff.org/files/2014/01/06/20131230-appelbaum-nsa_ant_catalog.pdf.

[44] Fundamentals, M. 2016. Payload types in the metasploit framework. https://www.offensive-security.com/metasploit-unleashed/payload-types/.

[45] Sands, M. B. 2015. Naming and identification of malware. https://www.rsaconference.com/writable/presentations/file_upload/tta-f04-high-dimensional-visualization-of-malware-families.pdf.

[46] Wu Zhou, Junyuan Zeng, L. S. J. S. 2016. A growing number of android malware families believed to have a common origin: A study based on binary code. https://www.fireeye.com/blog/threat-research/2016/03/android-malware-family-origins.html.

[47] Ero Carrera, P. S. 2010. State of malware: Family ties. https://media.blackhat.com/bh-eu-10/presentations/Carrera_Silberman/BlackHat-EU-2010-Carrera-Silberman-State-of-Malware-slides.pdf.

[48] Wikipedia, malware analysis wiki site, 2016. https://en.wikipedia.org/wiki/Malware_analysis. Accessed: 2016-08-21.

[49] Chris pope, "quick look: “kins” and mag2, blue coat lab", 2013. https://www.bluecoat.com/security-blog/2013-07-26/quick-look-%E2%80%9Ckins%E2%80%9D-mag2. Accessed: 2016-08-21.

[50] Wikipedia. 2016. Reverse engineering. https://en.wikipedia.org/wiki/Reverse_engineering.

[51] Tully, J. 2008. Print introduction into windows anti-debugging. https://www.codeproject.com/articles/29469/introduction-into-windows-anti-debugging.

[52] Unknown. January 2005. Pe file structure. http://repo.hackerzvoice.net/depot_madchat/vxdevl/papers/winsys/pefile/pefile.htm.

[53] Pietrek, M. March 1992. Peering inside the pe: A tour of the win32 portable executable file format. https://msdn.microsoft.com/en-us/library/ms809762.aspx.

[54] Pietrek, M. February 2002. An in-depth look into the win32 portable executable file format. *MSDN Magazine*. https://msdn.microsoft.com/en-us/magazine/cc301805.aspx.

[55] Kath, R. January 1997. The portable executable file format from top to bottom. http://www.csn.ul.ie/ caolan/pub/winresdump/winresdump/-doc/pefile2.html.

[56] Tell, M. January unknown. Pe files, an article describing pe files. http://www.silurian.com/inspect/peformat.htm.

[57] organization, S. 2004. Portable formats specification, version 1.1. http://www.skyfree.org/linux/references/ELF_Format.pdf.

[58] Michael Bailey, J. O. 2013. Automated classification and analysis of internet malware. https://web.eecs.umich.edu/ zmao/Papers/raid07_final.pdf.

[59] Agobot, wikipedia page abouth agobot, 2016. https://en.wikipedia.org/wiki/Agobot. Accessed: 2016-05-10.

[60] Mihai Christodorescu. Somesh Jha, S. A. S. 2013. Semantics-aware malware detection, berkely unirversity. http://www.eecs.berkeley.edu/ sseshia/pubdir/oakland05.pdf.

[61] Debin Gao, Michael K. Reiter, D. S. 2013. Behavioral distance measurement using hidden markov models. http://www.cs.berkeley.edu/ dawnsong/papers/hmm-raid06.pdf.

[62] Tian, R. 2011. An integrated malware detection and classification system. https://dro.deakin.edu.au/eserv/DU:30043244/Tian-thesis-2011.pdf.

[63] Younghee Park, Douglas Reeves, V. M. 2010. Fast malware classification by automated behavioral graph matching. http://modusoperandi.csc.ncsu.edu/papers/graphmatch.pdf.

[64] Carrera, E. & Erdélyi, G. 2004. Digital genome mapping – advanced binary malware analysis. http://bigdaddy.fr/repository/Documentation/Hacking/Security/Malware/Digital

[65] Hex-ray corp, ida: About information site, 2016. `https://www.hex-rays.com/products/ida/`. Accessed: 2016-10-31.

[66] Hex-ray corp, ida python api, 2016. `https://www.hex-rays.com/products/ida/support/idapython_docs/`. Accessed: 2016-10-31.

[67] Briones, I. & Gomez, A. 2008. Graphs, entropy and grid computing: Automatic comparison of malware. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.462.7386 &rep=rep1&type=pdf.

[68] Jusuk Lee, Kyoochang Jeong, H. L. 2010. Detecting metamorphic malwares using code graphs. http://dl.acm.org/citation.cfm?id=1774505.

[69] Ming Xu, Lingfei Wu, e. 2013. A similarity metric method of obfuscated malware using function-call graph. http://dl.acm.org/citation.cfm?id=2447439.

[70] Domagoj Babic, D. R. & Song, D. N/A. Malware analysis with tree automata inference. http://www.cs.berkeley.edu/ dawnsong/papers/201120cav11malware.pdf.

[71] Matt Fredrikson, Somesh Jha, e. 2010. Synthesizing near-optimal malware specifications from suspicious behaviors. *IEEE*. https://www.cs.ucsb.edu/ xyan/papers/oakland20_malware.pdf.

[72] Mihai Christodorescu, Somesh Jha, C. K. 2007. Mining specifications of malicious behavior. *ACM*. http://dl.acm.org/citation.cfm?id=1287628.

[73] Stavros D. Nikolopoulos, I. P. Detecting malicious code by exploiting dependencies of system-call groups.

[74] Sand, information-based dependency matching a study of information-based dependencies of function calls, 2013. http://www.sikkerhetsfokus.no/wp-content/uploads/2011/08/Information-based-dependency-matching-v1.pdf. Accessed: 2017-04-27.

[75] Yuping Li, Sathya Sundaramurthy, e. 2015. Experimental study of fuzzy hashing in malware clustering analysis. https://www.usenix.org/system/files/conference/cset15/cset15-li.pdf.

[76] Wikipedia, md5 hash wiki site, 2016. https://en.wikipedia.org/wiki/MD5. Accessed: 2016-05-10.

[77] Wikipedia, sha 256 hash wiki site 2016. https://en.wikipedia.org/wiki/SHA-2. Accessed: 2016-05-10.

[78] Shaddowserver crew, shaddowserver, 2014. https://www.shadowserver.org/wiki/. Accessed: 2016-05-10.

[79] Virus total corp, virustotal information site, 2016. https://www.virustotal.com/nb/about/. Accessed: 2016-05-10.

[80] Sourceforge, ssdeep information, 2013. http://ssdeep.sourceforge.net/usage.html. Accessed: 2016-05-10.

[81] Tanembaum, A. 2009. *Modern operating system, book release*. Pearson, third edition edition, https://www.amazon.com/Modern-Operating-Systems-Andrew-Tanenbaum/dp/013359162X.

[82] Wikipedia, mcrosoft files desciption, ntdll.dll, 2015. https://en.wikipedia.org/wiki/Microsoft_Windows_library_files#NTDLL.DLL. Accessed: 2016-05-11.

[83] Ryder, B. 2006. Constructing the call graph of a program. http://ieeexplore.ieee.org/document/1702621/.

[84] Xin Hu, Tzi Chiueh, K. S. 2009. Large-scale malware indexing using function-call graphs. http://www.ecsl.cs.sunysb.edu/tr/TR246.pdf.

[85] Bengoetxea, E. 2002. Inexact graph matching using estimation of distribution algorithms. http://www.sc.ehu.es/acwbecae/ikerkuntza/these/thesis.pdf.

[86] Dania Koutra, Ankur, e. 2011. Algorithms for graph similarity and subgraph matching. https://www.cs.cmu.edu/ jingx/docs/DBreport.pdf.

[87] Dekker, R. 2006. The importance of having data-sets. http://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1760&context=iatul.

[88] Darpa intrusion detection evaluation group. 1998. 1998 darpa intrusion detection evaluation data set. `http://www.ll.mit.edu/mission/communications/its/corpora/ideval/data/1998data.html`. Accessed: 2016-11-07.

[89] Darpa intrusion detection evaluation group. 1999. 1999 darpa intrusion detection evaluation data set. `http://www.ll.mit.edu/mission/communications/its/corpora/ideval/data/1999data.html`. Accessed: 2016-11-07.

[90] Darpa intrusion detection evaluation group. 2000. 2000 darpa intrusion detection evaluation data set. `http://www.ll.mit.edu/mission/communications/its/corpora/ideval/data/2000data.html`. Accessed: 2016-11-07.

[91] Dataset for anti-malware research and research achievements shared at the workshop. `http://www.iwsec.org/mws/2009/paper/A1-1.pdf`. Accessed: 2016-11-07.

[92] M. hatada y. nakatsuru, m. terada s.yoichi ,anti-malware engineering workshop 2008. `http://www.iwsec.org/mws/2008/en.html`. Accessed: 2016-11-07.

[93] Kaggle, microsoft malware classification challenge (big 2015), 2015. `https://www.kaggle.com/c/malware-classification`. Accessed: 2016-11-07.

[94] VirusShare. 2017. Virusshare.com - because sharing is caring. https://tracker.virusshare.com:7000/torrents/VirusShare_ELF_20140617.zip.torrent?FC417D1A9B39DBD33E260E6653A7BB194E974ED3.

[95] Oracle virtualbox, about virutalbox home page, 2015. `https://www.virtualbox.org/wiki/VirtualBox`. Accessed: 2016-11-07.

[96] Wikipedia-The-Free-encyclopedia. 2016. Wikipedia page 'tracing (software)'. https://en.wikipedia.org/wiki/Tracing_(software).

[97] Hex-ray corp, ida plugin contest 2015 hex-rays, 2015. https://hex-rays.com/contests/2015/. Accessed: 2016-10-31.

[98] Yaniv balmas, ida plugin die, 2015. https://github.com/ynvb/DIE. Accessed: 2016-10-31.

[99] Intel inc., intel® vtune^{TM} amplifier, 2017. https://software.intel.com/en-us/intel-vtune-amplifier-xe. Accessed: 2017-04-18.

[100] Garg, P. N/A. Stracent – system call tracer for windows nt. http://intellectualheaven.com/Articles/StraceNT.pdf.

[101] Linux-Manual-pages. 2017. ltrace(1) - linux man page. https://linux.die.net/man/1/ltrace.

[102] Linux pages, strace(1) - linux man page, 2016. https://linux.die.net/man/1/strace. Accessed: 2017-04-18.

[103] Funprogramming corp., function parameters, call and return value, 2015. http://funprogramming.org/31-Function-parameters-and-return-values.html. Accessed: 2016-10-31.

[104] Ntnu, andrii shalaginov information site. https://www.ntnu.no/ansatte/andrii.shalaginov. Accessed: 2016-05-10.

[105] Booker JM, M. L. 2004. *Solving black box computation problems using expert knowledge theory and methods*. Booker JM, 1. edition, http://www.sciencedirect.com/science/article/pii/S0951832004000705.

[106] Drescher, MA. Perera AH, B. L. e. a. . 2008. *Uncertainty in expert knowledge of forest succession: a case study from boreal Ontario*. For Chron, 1. edition, http://pubs.cif-ifc.org/doi/abs/10.5558/tfc84194-2.

[107] Ajith H. Perera, C. Ashton Drew, C. J. J. 2006. *Expert Knowledge and Its Application in Landscape Ecology*. Whiley Publishing Inc, 1. edition.

[108] Dharwadker, A. 2006. The vertex cover algorithm. http://www.dharwadker.org/vertex_cover/.

[109] Geeksforgeeks, vertex cover problem | set 1 (introduction and approximate algorithm), 2013. http://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm-2/. Accessed: 2017-04-27.

[110] Ammar Elhadi, M. M. May 2013. Improving the detection of malware behaviour using simplified data dependent api call graph. *International Journal of Security and Its Applications*, 7(5). http://docplayer.net/12935295-Improving-the-detection-of-malware-behaviour-using-simplified-data-dependent-api-call-graph.html.

[111] Neo4j project, intro to cypher, 2017. `https://neo4j.com/developer/cypher-query-language/`. Accessed: 2017-03-10.

[112] Gephi-assosiation. 2017. Gephi makes graphs handy. https://gephi.org/features/.

[113] Dato labs, graphlab create, 2016. `https://dato.com/products/create/`. Accessed: 2016-05-10.

A Preprosessing script

This Appendix is the pre prosseing script used to convert a Ltrace or Strace output into a CSV file which is used for matching by the script in Appendix B. The script is written in Python and takes a trace output as a argument and write new output file as the result.

```
#!/usr/bin/python
import sys
#print 'Number of arguments:', len(sys.argv), 'arguments.'
#print 'Argument List:', str(sys.argv)
if "aself" in sys.argv[1]:
   family = "aself"
if "cornelgen" in sys.argv[1]:
   family = "cornelgen"
if "madvise" in sys.argv[1]:
   family = "madvise"
if "rst" in sys.argv[1]:
   family = "rst"
if "tsunami" in sys.argv[1]:
   family = "tsunami"

#init
counter=0
calls = []
callsProcessed = []
file = open (sys.argv[1],"r")

#read line into array
for line in file.readlines():
    # loop over the elements, split by whitespace
    calls.append(line)
final = []
finalList = []

#Preprosess list
for line in calls:
   callsProcessed.append(line.strip().split(' '))
```

```
#Append values to list
for line in callsProcessed:
   final.append(line[2].split('(')[0])
   #print line[3].split('(')[0]
   try:
      final.append(callsProcessed[counter][3].split('(')[0].replace(")",""))
   except:
      pass
   try:
      final.append(callsProcessed[counter][4])
   except:
      pass
   try:
      final.append(callsProcessed[counter][5])
   except:
      pass
   try:
      final.append(callsProcessed[counter][6])
   except:
      pass
   try:
      final.append(callsProcessed[counter][-1])
   except:
      pass
   #print final
   finalList.append(final)
   final = []
   counter=counter+1

nodeList = []
fullNodeList = [[]]
counter = 0
neoOutfile = sys.argv[1] + ".nodes.csv"
neoFile = open(neoOutfile, 'w+')

#neoFile.write(listStart1)
for item in finalList:
   if counter == len(finalList) - 1:
      nodeList.append(finalList[counter][0].replace(",","").replace("'","")
         + ',' + finalList[counter][0].replace(",","").replace("'","") +
         ",")
      nodeList.append(finalList[counter][1].replace(",","").replace("'","")
         + ",")
```

```
        nodeList.append(finalList[counter][2].replace(",","").replace("'","")
            + ",")
        nodeList.append(finalList[counter][3].replace(",","").replace("'","")
            + ",")
        nodeList.append(finalList[counter][-1] + ",")
        nodeList.append(family)
    else:
        nodeList.append(finalList[counter][0].replace(",","").replace("'","")
            + ',' + finalList[counter][0].replace(",","").replace("'","") +
            ",")
        nodeList.append(finalList[counter][1].replace(",","").replace("'","")
            + ",")
        nodeList.append(finalList[counter][2].replace(",","").replace("'","")
            + ",")
        nodeList.append(finalList[counter][3].replace(",","").replace("'","")
            + ",")
        nodeList.append(finalList[counter][-1] + ",")
        nodeList.append(family)
    for item in nodeList:
        neoFile.write(str(item))
    neoFile.write("\n")
    fullNodeList.append(nodeList)
    nodeList = []
    counter = counter + 1

neoOutfile = sys.argv[1] + ".edges.csv"
neoFile = open(neoOutfile, 'w+')
#neoFile.write("Source,Target,Weight\n")
weightList = [[]] * len(fullNodeList)

# Argument value rules implementation
# 1. No single characters
# 2. No integers
# 3. Strings are unreliable, except from memory-values and and handles
#        - Only 0x strings are valid
# 4. Ordering dependencies to avoid false dependencies
#    - Value must be created by previous call, not subsequent call
intString = "int"
typeString = ""
counter = 0
for item in fullNodeList:
    try:
        value = fullNodeList[counter-1][-2]
        typeString = str(type(value))
    except:
        weightList[counter] = 1
    try:
        if len(value) == 2:
            weightList[counter] = 1
        elif intString in typeString:
            weightList[counter] = 1
    except:
        pass
    try:
        if fullNodeList[counter-1][-2] in fullNodeList[counter][1:4]:
            weightList[counter] = 5
        else:
            weightList[counter] = 1
    except:
        pass
    counter = counter + 1

edgeList = []
counter = 0

for item in finalList:
    if counter == len(finalList) - 1:
        pass
    elif counter == len(finalList) - 2:
        edgeList.append(finalList[counter][0].replace(",","").replace("'","")
            + ',' + finalList[counter+1][0].replace(",","").replace("'","")
            + "," + str(weightList[counter]))
    else:
        edgeList.append(finalList[counter][0].replace(",","").replace("'","")
            + ',' + finalList[counter+1][0].replace(",","").replace("'","")
            + "," + str(weightList[counter]))
    for item in edgeList:
        neoFile.write(str(item))
        neoFile.write("\n")
    edgeList = []
    counter = counter + 1
```

B Matching script

This Appendix is the matching script used in this thesis. This is a plain text implementation of the inexact graph matching algorithm described in Section 6.3.1. This script takes the output of the preprocessing script in Appendix A and conducts the matching. The script takes a CSV file as input and match it against the derived expert graphs, calculates the match ratio and classifies the input file. The results are printed to stdout. The expert graphs can be downloaded at: 188.166.102.216/assets/ExpertGraphs.7z with "infected" as the password for the downloaded zip file.

```
import sys
import operator

def matchNodeSimple(expert):
   counter = 0
   file1Calls = []
   file2Calls = []
   callsSimple1 = []
   callsSimple2 = []

   #Open and Preprosess file1
   with open(sys.argv[1], 'r') as file1:
      for line in file1.readlines():
         file1Calls.append(line)
      for line in file1Calls:
         callsSimple1.append(line.strip().split(','))

   #Open and Preprosess file1
   with open(expert, 'r') as file2:
      for line in file2.readlines():
         file2Calls.append(line)
      for line in file2Calls:
         callsSimple2.append(line.strip().split(','))

   #Match lists
   ratioCounter = 0
   for line in callsSimple1:
      if line[0:2] in callsSimple2:
            ratioCounter = ratioCounter + 1
    expert = getFamily(expert)
    if len(callsSimple1) <= 40:
        print "To few call nodes, cant not match"
        exit()
    else:
        #print "Match ratio: " +
            str(float(ratioCounter)/float(len(callsSimple1))) + "\t\t" +
            expert
        return ratioCounter

def matchNodeFull(expert):
    counter = 0
    file1Calls = []
    file2Calls = []
    callsSimple1 = []
    callsSimple2 = []

    #Open and Preprosess file1
    with open(sys.argv[1], 'r') as file1:
        for line in file1.readlines():
            file1Calls.append(line)
        for line in file1Calls:
            callsSimple1.append(line.strip().split(','))

    #Open and Preprosess file1
    with open(expert, 'r') as file2:
        for line in file2.readlines():
            file2Calls.append(line)
        for line in file2Calls:
            callsSimple2.append(line.strip().split(','))

    #Match lists
    ratioCounter = 0
    for line in callsSimple1:
        #print callsSimple2
        if line[0:6] in callsSimple2:
            ratioCounter = ratioCounter + 1
    expert = getFamily(expert)
    if len(callsSimple1) <= 20:
        print "To few call nodes, cant not match"
        exit()
    else:
      #print "Match ratio: " +
          str(float(ratioCounter)/float(len(callsSimple1))) + "\t\t" +
          expert
      return ratioCounter

def matchEdgeFull(expert):
   counter = 0
   file1Calls = []
   file2Calls = []
   callsSimple1 = []
   callsSimple2 = []

   #Open and Preprosess file1
   with open(sys.argv[1], 'r') as file1:
      for line in file1.readlines():
         file1Calls.append(line)
      for line in file1Calls:
         callsSimple1.append(line.strip().split(','))

   #Open and Preprosess file1
   with open(expert, 'r') as file2:
      for line in file2.readlines():
         file2Calls.append(line)
      for line in file2Calls:
         callsSimple2.append(line.strip().split(','))

   #Match lists
   ratioCounter = 0
   for line in callsSimple1:
      if line in callsSimple2:
         ratioCounter = ratioCounter + 1
   expert = getFamily(expert)
   if len(callsSimple1) <= 20:
      print "To few call nodes, cant not match"
      exit()
   else:
      #print "Match ratio: " +
          str(float(ratioCounter)/float(len(callsSimple1))) + "\t\t" +
          expert
      return ratioCounter

def getFamily(fam):
   family = ""
```

```
    if "aselF" in fam:
        family = "Aself"
    if "cornelgen" in fam:
        family = "Cornelgen"
    if "madvise" in fam:
        family = "Madvise"
    if "rst" in fam:
        family = "Rst"
    if "tsunami" in fam:
        family = "Tsunami"
    return family

if __name__ == "__main__":
    usage = "Usage: python match.py <trace>"

    if len(sys.argv) != 2:
        print "[ERROR]: Not enough arguments\n" + usage
        exit()

    print sys.argv[1]
    fam = getFamily(sys.argv[1])
    classificationNS = []
    classificationEF = []

    classification =[]
    familiyClass = ""
    #Match with expertfile
    #print "+++ Match nodeSimple +++"
    aselfCount = matchNodeSimple("/home/ltrace/nodeSimple/aself.expert")
    cornelgenCount =
        matchNodeSimple("/home/ltrace/nodeSimple/cornelgen.expert")
    madviseCount =
        matchNodeSimple("/home/ltrace/nodeSimple/madvise.expert")
    rstCount = matchNodeSimple("/home/ltrace/nodeSimple/rst.expert")
    tsunamiCount =
        matchNodeSimple("/home/ltrace/nodeSimple/tsunami.expert")
    classificationNS.extend((aselfCount, cornelgenCount, madviseCount,
        rstCount, tsunamiCount))

    #print "+++ Match edgeFull +++"
    aselfCount = matchNodeFull("/home/ltrace/edgeFull/aself.expert")
    cornelgenCount =
        matchNodeFull("/home/ltrace/edgeFull/cornelgen.expert")
```

```
madviseCount = matchNodeFull("/home/ltrace/edgeFull/madvise.expert")
rstCount = matchNodeFull("/home/ltrace/edgeFull/rst.expert")
tsunamiCount = matchNodeFull("/home/ltrace/edgeFull/tsunami.expert")
classificationEF.extend((aselfCount, cornelgenCount, madviseCount,
    rstCount, tsunamiCount))

if classificationNS.count(0) == len(classificationNS) and
    classificationEF.count(0) == len(classificationEF):
   print "No matches for trace"
   exit()
if max(classificationNS) == 0:
   pass
else:
   classification.append(classificationNS.index(max(classificationNS))+1)
if max(classificationEF) == 0:
   pass
else:
   classification.append(classificationEF.index(max(classificationEF))+1)
   classification.append(classificationEF.index(max(classificationEF))+1)
none = classification.count(0)
aself = classification.count(1)
cornelgen = classification.count(2)
madvise = classification.count(3)
rst = classification.count(4)
tsunami = classification.count(5)
if aself > cornelgen and aself > madvise and aself > rst and aself >
    tsunami:
   #print "Sample has been classified as Aself"
   familiyClass = "Aself"
if cornelgen > aself and cornelgen > madvise and cornelgen > rst and
    cornelgen > tsunami:
   #print "Sample has been classified as Cornelgen"
   familiyClass = "Cornelgen"
if madvise > cornelgen and madvise > aself and madvise > rst and
    madvise > tsunami:
   #print "Sample has been classified as Madvise"
   familiyClass = "Madvise"
if rst > cornelgen and rst > madvise and rst > aself and rst > tsunami:
   #print "Sampl ec has been classified as rst"
   familiyClass = "Rst"
if tsunami > cornelgen and tsunami > madvise and tsunami > rst and
    tsunami > aself:
   #print "Sample has been classified as Tsunami"
   familiyClass = "Tsunami"

#Matching stats
print "Nodes: " + str(classificationNS)
print "Edges: " + str(classificationEF)
print "Class: " + str(classification)
print "Values: " + str(aself) + str(cornelgen) + str(madvise) +
    str(rst) + str(tsunami)
print str(fam) + " Classifed as: " + str(familiyClass)
if fam == familiyClass:
   print "CORECT!!"
else:
   print "FAILED!!"
```

C Ida python script

This Appendix is the work done for implementing call graph extraction for PE32 files with IDA pro and IDAPython. This script is used inside IDA and utilizes the DIE plugin for IDA pro. The results of this script is a call graph file viewable in WinGraph32, which is the programs used in Section 6.2.6.

```
import os
from time import ctime
import sys
import time
from idaapi import plugin_t
import idaapi
import idautils
import idc
import sark.ui
from DIE.Lib import DebugAPI
from DIE.Lib.IDAConnector import *
import DIE.Lib.DieConfig
import DIE.Lib.DIEDb
from idaapi import PluginForm
from itertools import izip

#Wait for Ida auto analysis
idaapi.autoWait()

#Start debugging from start to end
debugAPI = DebugAPI.DebugHooker(is_dbg_pause=False, is_dbg_profile=False)
debugAPI.start_debug(None, None, auto_start=True)

die_db = DIE.Lib.DIEDb.get_db()
#Extract call graph list with values
call_graph_list = []
prev_func_ea = None
next_func_ea = None
counter = 0
call_value = []
return_value = []
for cur_context_id in die_db.function_contexts:
   cur_context = die_db.function_contexts[cur_context_id]
   for call_value_id in cur_context.call_values:
       dbg_value = die_db.dbg_values[call_value_id]
       parsedCall_value = die_db.get_parsed_values(dbg_value)
       for value in parsedCall_value:
           call_value.append(value.data)
   for ret_value_id in cur_context.ret_values:
       dbg_value = die_db.dbg_values[ret_value_id]
       parsedRet_value = die_db.get_parsed_values(dbg_value)
       for ret_value in parsedCall_value:
           return_value.append(ret_value.data)
   if cur_context.function in die_db.functions:
       prev_func_ea =
           die_db.functions[cur_context.function].function_start
       prev_func_name =
           die_db.functions[cur_context.function].function_name
   for next_context_id in cur_context.child_func_ctxt_id_list:
       if next_context_id in die_db.function_contexts:
           next_context = die_db.function_contexts[next_context_id]
           if next_context.function in die_db.functions:
               next_func_ea =
                  die_db.functions[next_context.function].function_start
               next_func_name =
                  die_db.functions[next_context.function].function_name
       if prev_func_ea and next_func_ea:
           call_graph_list.append((counter,prev_func_name,
               call_value,next_func_name, return_value, prev_func_ea,
               next_func_ea))
           counter = counter + 1
           call_value = []
           return_value = []
print call_graph_list

#Create GDL file from call_graph_list
callGraph = call_graph_list
functions = []
counter = 0
for i in callGraph:
   functions.append(callGraph[counter][1])
   functions.append(callGraph[counter][3])
   counter = counter + 1
functions = list(set(functions))
node = []
```

```
counter = 0
for i in functions:
    temp = "node: { title: \"" + str(functions[counter]) + "\" label: \""
        + str(functions[counter]) + "\" }"
    node.append(temp)
    counter = counter + 1
edge = []
counter = 0
for i in callGraph:
    callValue = str(callGraph[counter][2]).replace("\"", "")
    temp = "edge: { sourcename: \"" + str(callGraph[counter][1]) + "\"
        targetname: \"" + str(callGraph[counter][3]) + "\" label: \"" +
        str(callValue) + "\" }"
    edge.append(temp)
    counter = counter + 1
thefile = open('test', 'w')
thefile.write("graph: {\ntitle: \"Title\"\n")
for item in node:
    thefile.write("%s\n" % item)
thefile.close()
thefile = open('test', 'a')
for item in edge:
    thefile.write("%s\n" % item)
thefile.write("}")
thefile.close()

#idc.Exit(0)
```

D Bash script virtual for environment

This Appendix presents two bash scripts for used to batch run the samples in a virtual environment. The first script is used to start, load files, execute a script on the guest machine, extract logs ans shut down the guest machine in VirutalBox. The second script is the script executed by the first, which loads a malware and trace it with either Ltrace or Strace.

Script for running virtual environment

```
#!/bin/bash
counter=1
##Loop for loading new malware to malwareFolder
for file in test/*; do
   echo "Copying file $file..."
   cp $file /home/user/Documents/malwareFolder/

   ##StartVM
   VBoxManage startvm "Kubuntu" --type headless
   sleep 45
   ##Run analysis script
   timeout 40 VBoxManage guestcontrol "Kubuntu" run --exe
       /home/user/Documents/doIt.sh --username user --password user

   ##ShutdownVM
   VBoxManage controlvm "Kubuntu" poweroff
   sleep 5
   ##Revert to clean snapshot
   VBoxManage snapshot "Kubuntu" restore "time3"
   sleep 5

   let counter=counter+1
   rm /home/user/Documents/malwareFolder/*
   echo "Finished analysing number: $counter/500"
done
```

Script for running ltrace and strace inside virtual host

```
#!/bin/bash
```

```
type='ltrace'
malwarePath='/tmp/malwareFolder'
sshpass -p "user" scp -r
   user@192.168.1.108:/home/user/Documents/malwareFolder/* $malwarePath

file="$(ls $malwarePath | head -1)"
echo "++++Starting" $type "Sctipt++++"
echo "Malware to be traced: " $file
echo "Log file: " $file.log
touch /tmp/$file.log
chmod +rwx $malwarePath/$file
echo "user" | sudo -S $type -o /tmp/$file.$type -ttt -f
   $malwarePath/$file > /tmp/$file.log
echo "Tracing is done, tracefile:"
ls /tmp | grep $file.$type
echo "Copying files to External host machine"
echo "user" | sudo -S chown -R user:user /tmp/
sshpass -p "user" scp /tmp/$file.$type /tmp/$file.log
   user@192.168.1.108:/home/user/Documents/trace
echo "+++Done++++"
```

E Virus Total script

This Appendix presents a script used to submit the samples to Virus Total via the API. This was used in the context to classify the samples when generating the dataset.

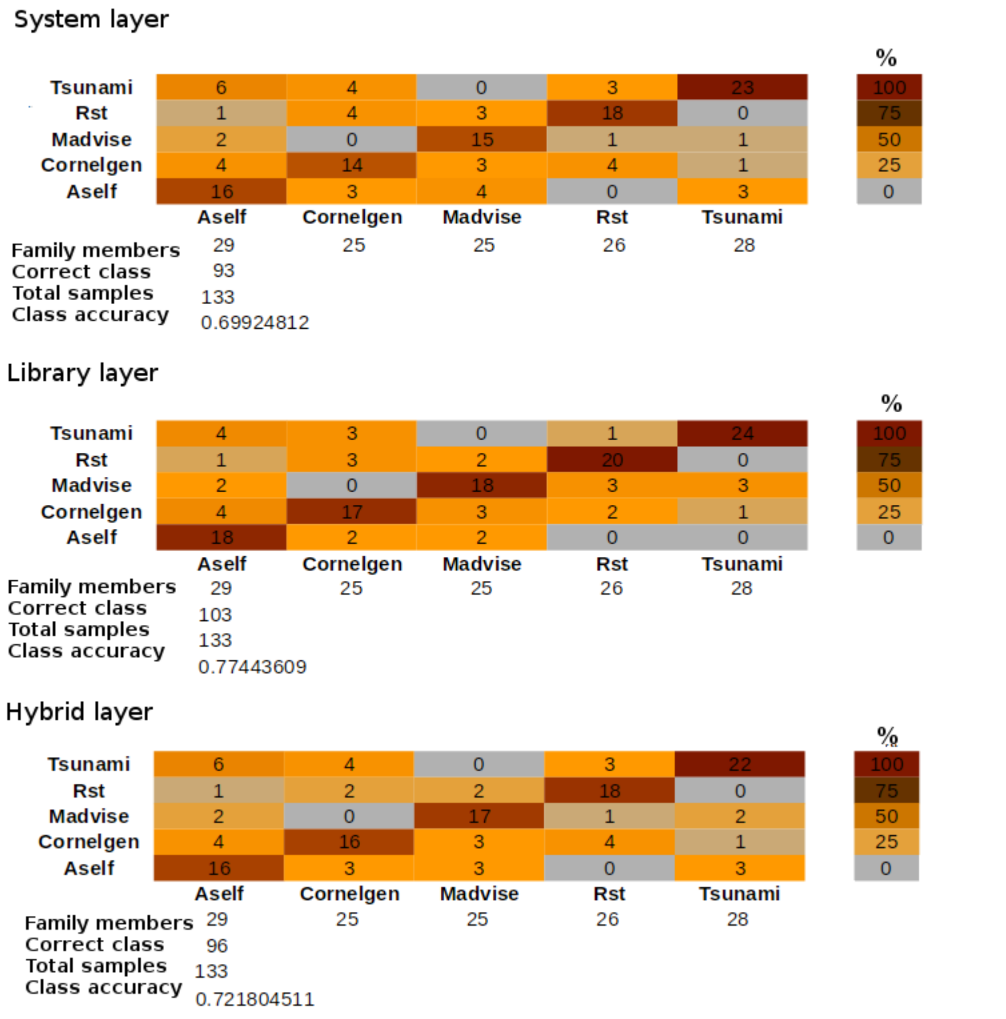
```
import requests
import sys

md5 = sys.argv[1]
print md5
params = {'apikey': 'APIKEY', 'resource': md5}
headers = {
  "Accept-Encoding": "gzip, deflate",
  "User-Agent" : "Matser thesis projcet, thx for shearing"
  }
response =
    requests.get('https://www.virustotal.com/vtapi/v2/file/report',
  params=params, headers=headers)
json_response = response.json()
json_response = str(json_response)

filename = md5 + '.txt'
file = open(filename,"w")
file.write(json_response)
file.close()
```

F Classifcitacion accuracy HeatMap

This Appendix presents the data for the heatmaps for the different layers of classification.



System layer

	Aself	Cornelgen	Madvise	Rst	Tsunami
Tsunami	6	4	0	3	23
Rst	1	4	3	18	0
Madvise	2	0	15	1	1
Cornelgen	4	14	3	4	1
Aself	16	3	4	0	3
Family members	29	25	25	26	28
Correct class	93				
Total samples	133				
Class accuracy	0.69924812				

%
100
75
50
25
0

Library layer

	Aself	Cornelgen	Madvise	Rst	Tsunami
Tsunami	4	3	0	1	24
Rst	1	3	2	20	0
Madvise	2	0	18	3	3
Cornelgen	4	17	3	2	1
Aself	18	2	2	0	0
Family members	29	25	25	26	28
Correct class	103				
Total samples	133				
Class accuracy	0.77443609				

%
100
75
50
25
0

Hybrid layer

	Aself	Cornelgen	Madvise	Rst	Tsunami
Tsunami	6	4	0	3	22
Rst	1	2	2	18	0
Madvise	2	0	17	1	2
Cornelgen	4	16	3	4	1
Aself	16	3	3	0	3
Family members	29	25	25	26	28
Correct class	96				
Total samples	133				
Class accuracy	0.721804511				

%
100
75
50
25
0

Figure 36: Total heatmap with additional data for all layers