Master's thesis

NTNU
Norwegian University of Science and Technology

Magnus Simonsen Håland

Multinomial malware classification using control flow graphs

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
June 2019

NTNU
Norwegian University of
Science and Technology

Magnus Simonsen Håland

Multinomial malware classification using control flow graphs

Master's thesis in Information Security
Supervisor: Geir Olav Dyrkolbotn
June 2019

Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Multinomial malware classification using control flow graphs

Magnus Simonsen Håland

01-06-2019

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Information Security and Communication Technology
Norwegian University of Science and Technology,

Supervisor: Associate Professor Geir Olav Dyrkolbotn

Abstract

In the days of personal computers and the Internet, malicious software (malware) has become an increasing threat. The number of incidents involving malware, causing significant damages to both private and governmental institutions, has increased over the years. Malware developers constantly develop new malware, or extend existing malware with new functionality or obfuscation techniques. This makes keeping up with new malware an impossible task for malware analysts and incident response personnel, making computer assisted solutions a necessity.

Most methods for computer assisted malware classification based on static analysis fail to capture behavioural information, while methods based on dynamic analysis are often resource intensive and difficult to implement. This thesis proposes a method for multinomial malware classification based on static analysis, that includes the behavioural information of malware. This information is included by recovering control flow graphs from malware samples, and converting them into feature vectors that include both the behavioural information, and the information within the nodes. Several different implementations of the method is suggested in this thesis, and the performance of a sub-set of the possible implementations are evaluated in a ten-class classification problem.

Sammendrag

I disse dager med personlige datamaskiner og Internett har skadelig programvare (skadevare) blitt en økende trussel. Antall hendelser som involverer skadevare, som har forårsaker betydelige skader for både private og offentlige institusjoner, har økt gjennom årene. Skadevareutviklere utvikler stadig ny skadevare, eller utvider eksisterende skadevare med ny funksjonalitet eller nye teknikker for å vanskeliggjøre analyse. Dette gjør det å holde seg oppdatert på skadevare en umulig oppgave for skadevare-analytikere og hendelseshåndterings-personell, noe som gjør datamaskinassisterte løsninger en nødvendighet.

De fleste metoder for datamaskinassisterte skadevareklassifisering basert på statisk analyse mangler evnen til å inkludere atferdsmessig informasjon, og metoder som baserer seg på dynamisk analyse er som regel ressurskrevende og vanskelige å implementere. Denne oppgaven presenterer en metode for multinomisk klassifisering av skadevare basert på statisk analyse, som inkluderer atferdsmessig informasjon fra skadevaren. Denne informasjonen er inkludert ved å gjenopprette kontroll flyt grafer fra skadevaren, og konverter grafene til vektorer som inkluderer både atferdsmessig informasjon og informasjon fra nodene. Flere ulike implementasjoner av metoden er foreslått i oppgaven, og ytelsen til noen av de mulige implementasjonene er evaluert i ett ti-klasse klassifikasjons problem.

Acknowledgment

I would like to extend my sincere gratitude to my supervisor, Ass. Prof. Geir Olav Dyrkolbotn, for providing excellent guidance and support throughout this project. I would also like to thank Prof. Katrin Franke, for providing insight into specific topics related to machine learning. Additionally, I appreciated how my supervisor facilitated discussion with key people at NTNU for me as a distance student.

Secondly, I would like to thank the researchers at the Norwegian Defence Research Establishment and my colleagues for discussions and suggestions for improvements to the thesis.

Finally, I would like to thank my friends and family, and especially my girlfriend Synne Vegsund Lande, for support, motivation and patience throughout my studies.

Contents

List of Figures

List of Tables

List of Listings

Glossary

Adjacency matrix Method for representing graph structure as a matrix

angr Binary analysis framework for Python

Assembly instruction Machine code instruction disassembled to the Assembly language

Basic block Sequence of code without branching except for the entry and exit

Colors Fixed length abstraction of instructions in a basic block

Conditional jump Transfer of control flow to a specified address if a condition evaluates to true, otherwise proceed with next instruction

Control flow graph Directed graph representation of software

C A programming language

Depth first search Graph search algorithm

Disassembly Conbvert machnie code into the Assembly language

IDA Pro Interactive application for disassembling and debugging of compiled software

Malware classification Sort malware into different categories

Malware detection Filer malware from non-malicious, benign software

Malware Malicious software

Python A programming language

Scikit-learn Machine Learning framework in Python

Unconditional jump Transfer of control flow to a specified address

Sub-method abbreviations

CAM Color-based Adjacency Matrix (part 3 sub-method)

CAMc Color-based Adjacency Matrix with colors (part 3 sub-method)

DFSc Depth First Search colors (part 3 sub-method)

LN Limit nodes (part 2 sub-method)

MGM Multiple graphs mode (part 1 sub-method)

MLP Multilayer perceptron (part 4 sub-method)

NAMc Node-based Adjacency Matrix with colors (part 3 sub-method)

NB Naive Bayes (part 4 sub-method)

OF Ordering functions (part 2 sub-method)

RF Random Forrest (part 4 sub-method)

RL Removing library functions (part 2 sub-method)

SB Successor Based (part 3 sub-method)

SGM Single graph mode (part 1 sub-method)

SVM Support Vector Machine (part 4 sub-method)

1 Introduction

This chapter serves as an introduction to the project and its topics. It covers keywords and a description of the problems that the project attempts to solve, as well as justification and motivation behind the research, the research questions, and the planned contributions of the project.

1.1 Topics covered

The internet has enabled communications across the globe, allowing rapid exchange of information. Most people today rely on internet connected devices to do much of their daily activities, and many companies are critically dependent on the internet to conduct their operations. Some people see this reliance on technology as an opportunity for criminal activity, such as stealing private or confidential information, or to affect the operations of companies and nations. These people develop techniques and malicious software, malware for short, to attack and infect target computers, and use the internet to share and spread their knowledge. This has led to a rapid development of new attack tools and techniques.

To counter this, an industry has emerged around cyber security, with businesses specializing in detecting and handling incidents. This industry constantly develop new techniques for detecting and analyzing malware, to combat the constant changing landscape. However, due to the large number of files entering computer networks, it has become unfeasible for humans to analyze everything. Computer assisted methods that aim to detect and sort malware are therefore necessary as a triage. These methods enable incident responders to prioritize and spend their limited but valuable time focused on a sub-set of files with certain properties, rather than wasting their time on irrelevant files.

In this project, we study one such method for computer assisted malware sorting. This method performs multinomial classification of malware, with the goal of helping incident responders prioritize between different malware samples and make initial assessments if incidents.

1.2 Keywords

Control flow graph, static analysis, malware classification, multinomial classification, behavioural-based classification, machine learning.

1.3 Problem description

Malware is short for malicious software, and is a term used for any software that is designed to have a harmful, intrusive or unwanted effect on those who executes it. With the increase of complexity in software and the availability of complex and customizable malware, the amount of work required by a malware analyst to properly analyze all incoming malware exceeds what is possible by a human.

Additionally, malware developers continuously make new versions of their malware, adding or changing functionality or complexity.

There currently exists several methods for computer assisted malware classification, using both static and dynamic analysis techniques. Most static methods rely on signature matching, and are therefore weak at detecting new or obfuscated malware. The methods relying on dynamic techniques usually involve executing the malware in a controlled environment while observing it, trying to detect some behavioural signatures. These methods are better at discovering new malware or new versions of old malware, but are resource intensive and slow, as they need to set up a new controlled environment for every malware sample and execute them. Additionally, it is not always trivial to get the malware to execute, due to environmental dependencies and deliberate anti-analysis techniques.

Modelling software as graphs and using features of the graphs, or converting the graphs into feature vectors, to detect and classify malware is not new. However, most methods rely on function call graphs, where the nodes represent functions, recorded using dynamic analysis. Some methods rely on recovering the control flow graph of a program, where the nodes represent blocks of code and the edges represent jumps between them. These methods can be based on static analysis, removing the drawbacks of dynamic analysis while still capturing the behaviour of a malware through the structure of the graph, enabling them to detect new instances. Kruegel et al. successfully used this technique in [7] to detect computer worms from normal web traffic and benign software.

In our project, we aim to expand on the method presented by Kruegel et al. in [7], and use it to perform multinominal classification of malware. This enables our method to combine the efficiency of static methods, with the behavioural information of dynamic analysis. The goal of this thesis is to develop a new method for multinomial malware classification based on static analysis that includes behavioural information.

1.4 Justification motivation and benefits

Most computers and networks today are connected to the internet. Companies and individuals rely on these to do most of their activities, and damages to their devices or theft of information can have critical financial and operational consequences. Additionally, as was shown in [10], attacks on critical infrastructure affects everyone who relies on it. These attacks often involve malware, unknowingly downloaded from the internet or receive through email.

One scenario in which automatic malware classification is necessary is when an incident team must quickly react to and assess the damages of an incident. In many cases, after detecting the malware infection, critical time is lost by analyzing the malware to determine the impact of the incident. Initial knowledge about the malwares capabilities, receive from automate systems, can save valuable time and enable incident response personnel to quickly perform the correct actions. Correctly classifying incoming malware enables analysts to prioritize their effort, and to assess potential damages.

One of the drawbacks of malware classification methods based on static analysis is that they are unable to include behavioural information. This type of information is usually only included by dynamic methods. In this thesis, we develop a method that counters this drawback. Our method is based on static analysis, but includes behavioural information, in order to combine the benefits of both static and dynamic analysis.

1.5 Research questions

The method proposed in this thesis consists of 4 parts, described in chapter 3, and suggests several ways each part can be implemented. The goal of this thesis is not to conduct an exhaustive evaluation of all possible combinations of implementations, but to evaluate a set of possible combinations as a proof-of-concept for the method.

Hypothesis:

We believe that control flow graphs recovered using static analysis contains behavioural information that is important for multinomial malware classification, and will therefore be useful in this context.

Research questions:

1. How can control flow graphs that include behavioural information be recovered from malware using static analysis?
2. How can graphs be pre-processed to improve the performance of the proposed method?
3. How can feature vectors that include behavioural information be constructed from the graphs?
4. Given a set of possible implementations of the proposed method, what is its performance with regards to classification accuracy?

1.6 Planned contributions

The primary contribution of this project will be a method for automatic malware classification based on control flow graphs recovered using static analysis. This will include a study of existing techniques for control flow graph recovery, how the graphs can be processed to improve classification accuracy, and what methods exist for classifying graphs that takes graph topology into consideration. Additionally, a framework for development and comparison of control flow graph-based classification methods is presented. Finally, a a short discussion of existing class-sets is included, to discover which once may offer useful information and how they affect classification.

1.7 Thesis outline

This section presents an overview of the thesis, as well as a short summary off each chapter.

Chapter 2: Theory
This chapter presents background knowledge and related work that is relevant to the topics of the thesis. The definition of malware used in this thesis is presented, as well as different ways to group malware. Following this, several topics related to malware are described: malware definition and categorizations, analysis methods, automatic analysis, detection, classification, and anti-analysis techniques. Finally, control flow graphs are described and how they can be recovered from malware,

as well as methods for classifying and matching graphs.

Chapter 3: Methodology
This chapter presents a theoretic overview of the method proposed in this thesis. It presents the structure of the method and the proposed framework for implementation and comparison, and suggests several sub-methods that can be used to implement it.

A dataset of malware will be used to train and evaluate the proposed method. This dataset will also be presented in this chapter.

Chapter 4: Experiment
This chapter presents the practical component of the thesis, the problems encountered during the implementation, and the experiments.

Chapter 5: Results
The results of the experiment are presented and compared in this chapter.

Chapter 6: Discussion
This chapter provides a discussion off the results, as well as the considerations and implications of the thesis, and a description of future research that may be conducted to further study the subjects presented.

Chapter 7: Conclusion
The final chapter presents a summary and conclusion of the thesis.

2 Background theory

Malware is short for the term malicious software. This chapter will present a definition of this term, and serve as an introduction to the malware domain. It includes a description of different malware categories and ways categorize them. Following this, the topic of malware analysis will be described, presenting different manual and automatic analysis techniques, and how these techniques can be employed to automatically detect and classify malware. Different anti-analysis methods are then presented. Finally, control flow graphs are described, as well as graph-based malware classification methods.

2.1 Definition

The definition of malware that will be used in this thesis is primarily based on the one given by Sikorski and Honig [11], who defines malware as "any software that causes harm to a user, computer or network..." . Many other definitions include harm towards computers and networks, but fails to include the user aspect. Malware is often used today to steal information or money from users, causing them financial harm, without damaging the computer [12][13]. To expand on this definition, we include files that are not executed by themselves, but are opened in or by other programs that may contain vulnerabilities that the file exploits to cause harm. Therefore, the definition used in this thesis is:

"Malware is defined as any software or file that is designed to cause harm towards a user or computer environment."

There exist several different ways to categorize malware. In this thesis, we will be using the definitions used by Microsoft [1], originally defined by the Computer Antivirus Research Organization (CARO) [14]. The major advantage of this naming scheme is that it contains as much information about a malware sample as possible, while remaining systematic. Each name consists of five components: type, family, platform, variant and suffix, where type describe what the malware does, family is how it does it, platform is the operative system that the malware is designed for, variant is a sequential numbering of distinct versions of a malware family, and suffix describes the filetype. An example of this is shown in figure 1. For the purpose of this work, we will in this thesis only focus on three of the components: types, families and platform. These components are described in detail in the following sections, as well as relevant examples within the different categorizations. Variant and suffix are not relevant in this context, as they doe not provide information that can be used in classification.

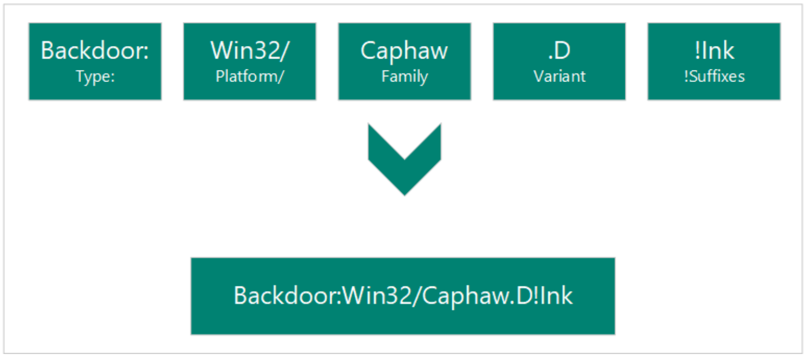


Figure 1: Example of a malware name, highlighting the different parts. From [1].

2.2 Malware types

Malware types groups malware by what they do on a computer or network. This may include how they infect computers, spread within networks and what they try to achieve after infecting a target. This section presents and discusses several different malware types. The list is not exhaustive, but covers the most commonly observed types.

2.2.1 Virus

Viruses are malware that spreads by attaching themselves to other files, and performs some action (which may be to only attach itself to other files) when the file is opened or executed [15]. By this definition, a virus requires a host file to spread. It does not have its own code for spreading between computers or networks, but relies on users sharing the infected file between them. Bishop [15] describes two phases of a successful virus infection: insertion and execution phase. During the first phase, a virus inserts itself into one or more target files. To avoid infecting already infected files, this phase often include a check to determine if a file has already been infected. During the second phase, the virus performs some, usually malicious, action. A piece of malware is therefore considered to be a virus by the way it spreads within a computer and between computers and networks, and not by the actions of the malware.

Viruses can be further divided into sub-categories, based on which files they attach themselves to. The most common categories are described below, together with a short description of polymorphism and metamorphism.

Boot sector virus

A boot sector virus attaches itself to the boot sector of a disk, and is executed with the boot sector. The boot sector of a disk contains that bootstraps the system, executing some code before passing execution to the operative system. To infect a systems boot sector, the virus must either copy itself into leftover space in the sector, or, more commonly, moving the original bootstrap code into another sector. In the lather case, the infected boot sector ends in a jump to the location where it moved the original boot section.

Executable virus

Executable viruses infect executable programs, and are executed when the host program is executed. The virus can inject its code anywhere in the executable file, as long as it modifies the programs execution flow in such a way that it jumps into the viruses' code at some point during execution. The most common location is at the executables entry point and moving the original code to some other place. This way, the viruses' code is executed before that programs code.

One drawback of this location is that executing the viruses' code may add a noticeable delay when starting the program, especially if the virus attempts to spread to several other files or perform some other computationally complex task.

Macro virus

Macro viruses attaches themselves to files or documents that are not executed, but interpreted by some other program. The virus code does therefore not consist of machine code, but instructions that are interpreted as part of the infected document when it is opened.

Polymorphism and metamorphism

Polymorphic and metamorphic viruses are those capable of altering themselves as they spread between files [15]. Polymorphic viruses are ones that changes their signature as they spread between files, by altering some part of its code without changing its functionality. Polymorphic viruses use these techniques to improve stealth. If the virus is discovered in one file, it is still difficult to find other infected files, as the virus signature will not match.

A common example of polymorphism are some encrypted viruses, where the main body of the virus is encrypted and only the decryption function and keys are visible. When infecting a new file, these viruses may decrypt their main body, and re-encrypt it using a new key. This new pair, along with the decryption function, is inserted into the new file. This way, the new infection has a signature different from the original, while achieving the same functionality. Some other viruses achieve this by changing their instructions into other instructions that produces the same result. Figure 2 presents three different assembly instructions all of which sets the value of register eax to zero.

Metamorphic viruses change both their signature and behaviour when they infect new files, in an attempt to evade both static and dynamic detection [16].

2.2.2 Worms

As with viruses, worms are defined by the way they spread between computers, and not by what they do after infection. Computer worms spread by copying themselves to other computers, usually without requiring any user interaction [11]. As opposed to viruses, they do not need to attach themselves to innocent files to execute. To spread, worms therefore require some way to replicate themselves across a network. This can be done using other programs or operative system features that enables copying files between computers, or by using some software exploit that enables the worm to copy itself.

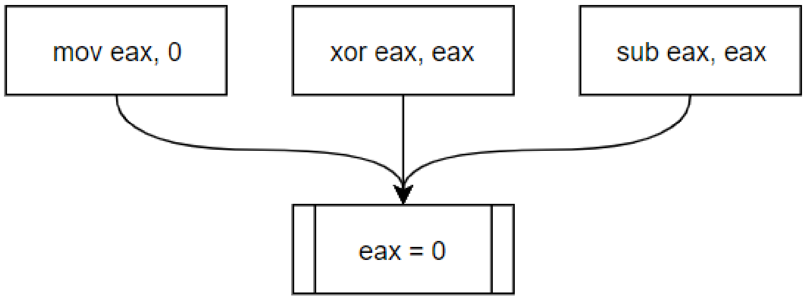


Figure 2: Examples of assembly instructions that can replace each other. All three eaxples sets the value of register eax to zero.

2.2.3 Trojan

A trojan is a type of malware that hides by masquerading as benign software. As with the previous two types, this one also only describes how the malware spreads, and not the actions it takes after infection. The name comes from the “trojan horse” myth of Greek mythology, where enemy soldiers invaded the city of Troy by hiding inside a large, wooden horse presented as a gift to the city. As in the myth, a computer user is tricked into installing the trojan by installing what they believe is benign software offering some useful functionality. The software often performs the advertised functionality as expected, while performing its malicious actions in the background [17], while other times the program may pretend to fail, after having installed its malicious payload. Either way, the user should not know that the malware is installed. To increase installation rate, the malware authors often hide their trojans as free software that performs some useful utility function, such as video compression or conversion [18].

2.2.4 Backdoor

Backdoors are defined by what they do, and now how they spread. After attacking a computer or network, malicious actors often leave backdoors that enables them to easily return [11]. This is necessary, because the original infection method may not be available at a later time, are may be difficult or costly to do again. The backdoor can be an application that the malicious actors install on the target computer, that allows them to remotely access the computer and execute malicious code. Backdoors can also reconfigure already installd software, e.g. create a new set of login credentials. The attacker can then use benign remote access tools, such as SSH or RDP to access the target computer. The backdoor type therefore covers both malicious applications that enables remote code execution, and code that re-configures other remote access software to bypass normal security measures for the attacker.

2.2.5 Botnet

Botnets are a form of backdoor that is installed on several computers, and usually controlled by a single source [11]. Each bot in the botnet usually receives the same commands from the source, and performs the same actions. The computer where the bot is installed is usually not the target of the attack, and is used either as a proxy for the attacker, or to increase the effect of an attack. Botnets are most commonly used in Dedicated Distributed of Service attacks (DDoS) [19], where individual infected hosts may not notice their participation, but the total traffic generated by all the bots in the network can be devastating to the target [20].

2.2.6 Rootkit

Rootkit describes malware that hides itself by manipulating functionality on the infected system. The name comes from the fact that rootkits usually requires elevated privileges to alter operative system functionality, and "root" is the name of the user account with the highest privileges on most Linux systems [21]. A rootkit may hide files or directories, processes running on the computer or network traffic, etc. The actions taken by the rootkit using elevated privileges cannot be audited by normal users, making it difficult for the user to detect rootkits after infection. However, to infect a system, a rootkit must somehow obtain these elevated privileges, either by obtaining valid credentials, or by exploiting some vulnerability in the operative system [22]. This makes rootkits more difficult to develop and use, and less common in the wild. Rootkits can further be divided into four categories, depending on what layer of the operative system they reside.

Application layer rootkits

Application layer rootkits are considered to be a type of rootkit, but does not require elevated privileges above that of a normal user.

This type of rootkit modifies user programs or commands to suppress information, which can be done, for instance, by creating or changing aliases in the command line to change which programs are executed. To list files in a directory on the Linux commandline, a user uses the "ls" command [23]. A rootkit can create an alias "ls" that points to a different program which executes the original ls command, but removes any files in the output that starts with a specific and uncommon prefix. By naming all its files with this prefix, the rootkit has effectively hidden itself from normal detect. This is simplified, as there are many other ways to find files that do not involve the "ls" command, but it serves to show how rootkits can hide themselves without requiring elevated privileges. Application layer rootkits are however easily discovered by users with elveated privileges, as they cannot modify commands used by those users.

Library layer rootkits

Library level rootkits hide by modifying the code of libraries that is called from other programs [24]. The rootkit can for instance modify functions for accessing and reading files, making them change the output whenever one of the rootkits files is accessed. This may also affect programs that a user may use to search for a rootkit if they suspect one has infected their computer, making these rootkits hard to find without specialized tools. Usually, shared libraries requires elevated privileges to change, making this type of rootkit more difficult to deploy.

Kernel layer rootkits

Kernel layer rootkits modify the operative system kernel to stay hidden [25]. The highest level of privileges on a computer is required to install this type of rootkit, and they require immense knowledge of the operative system kernel to develop, as even minor bugs may cause the computer to crash. Therefore, these are less common that the previous types. These rootkits are very difficult to detect, as it is no longer possible to trust the security measures of the operative system. Any tool that attempts to audit the system for malicious software or activities will receive reports modified by the rootkit, that states that everything is normal.

2.2.7 Remote Access Tool (RAT)

Remote Access Tools, or sometimes referred to as Remote Administration Tools, are a form of backdoor that offers more functionality that simple code execution [26]. There exists several examples of benign software that offers similar functionality, such as TeamViewer [27] and Windows Remote Desktop [28]. RATs are common types of malware, as they enable the attacker to performed several different actions after infecting a target, such as uploading or downloading files, keylogging, recording the screen or web-camera, etc. Some RATs also supports uploading files to the target computer, enabling the attacker to deploy additional malware and tools to infect other computers in a local network. RATs can also be used for recognizance, before selecting the right tools for the desired effect on the target.

2.2.8 Spyware

Spyware is designed to, as the name implies, to spy on the target user or computer, and gather information [11]. These may collect information about files, internet history or keyboard presses, or anything else that may be of interest to the attacker. The information is usually automatically uploaded to the attacker. This type of malware is often employed by criminals who wish to obtain banking information or credentials of targets.

2.2.9 Adware

Adware is a type of malware whose primary purpose is to display advertisement on the target computer [29]. Developers are payed by advertisement companies to develop software that contain adds, and are payed by the number of adds displayed. Therefore, by developing and infecting computers with malware that display these adds, they increase profit. Additionally, some adware alters the adds that the users see on normal webpages, making detection more difficult while still making money for the developer.

2.2.10 Ransomware

Ransomware is a newer type of malware that has been an emerging trend over the last few years, but has been a known phenomenon since 2005 [30]. This type of malware is designed to, after infecting a target computer, take control over some resource on the target, and demand that the user pays a ransom to the attacker. The most common method to do this is to encrypt several of the targets files and delete the local copy of the encryption key afterwards. The attacker can then send the key to the user after the ransom have been payed. Millions of dollars have been payed over the last few years in ransom [31][32], but the advice given by most security experts is not to pay, as there is no guaranty that the files will be decrypted afterwards.

2.2.11 Multiple types

Any given malware sample may belong in more than one of these categories, as it is common to include functionality from more that one type when developing malware. For instance, a rootkit may also be a RAT, and a botnet may spread like a virus. This adds an additional layer of complexity when grouping malware. A solution that is often used is to dived the malware into multiple components, where each component fits into a single type.

2.3 Malware families

Malware families group malware by how they do what they do [1]. As previously stated, the type of a malware describes what it does, while the family describes how it does it. How names are assigned to families varies from case to case. Usually, the first group of researchers that discovers a new family gets to decide the name. Often, strings found inside samples are used when assigning names [33]. There are also several examples where different security software providers assign different names to the same family.

A family refers to a collection of different versions or variants of a malware samples. A new sample is not referred to as a family on its own, but as different versions of it is discovered, a family is created to group them. This helps diversify them from other malware of the same type, which is useful when prioritizing between samples. New versions of malware are created in the same way new versions of other software is, usually by adding functionality or fixing errors, while new variants are created when malware is redesigned for different platforms or configured differently. Additionally, different malware samples attributed to the same author may be included in the same family.

By definition, malware of the same family usually belong to the same type. However, this is not always the case, as a new version of a malware sample may add additional functionality from another type. By assigning a family to malware, in addition to the type, we are able to further group similar malware, offering researchers more information about new samples.

As opposed to types, there does not exist a predefined list of malware families [34]. A new family is created whenever a group of samples is observed that does not fit within an existing family. We will therefore only present two malware families as examples. For additional examples, see [35] and [36].

2.3.1 Zeus

The Zues malware family is a collection of malware with functionality from the types spyware, trojan and botnet [37][38]. Samples within this family have primarily been used for criminal activity, targeting banks and users of online banking to steal login credentials. In 2011, the source code of one of the versions of Zeus was made public, resulting in a large number of different variants created by different authors.

2.3.2 Poison Ivy

Poison Ivy is a malware family of RATs that offer a large number of different features, including: keylogging, screen and camera recording, and file transfer [35]. Samples within this family were freely available online, up until 2008 with the final release of version 2.3.2. Due to its easy availability, this family has been used a lot over the year, primarily in criminal activity. FireEye [35] have collected 194 different samples in the Poison Ivy family, used in attacks between 2008 and 2013. These samples had a great number of different configurations, including different passwords and command and control information.

2.4 Platform

The platform component of the malware names indicates which platform or operative system the malware is designed to run on [1]. Additionally, the programming language and filetype is sometimes included, if, for instance, the malware is written in an interpreted language that can be executed on multiple platforms. There are several different platforms, too many to describe in this thesis. We have therefore focused on the two most relevant platforms: Windows and Linux. The next sections describe these platforms and their primary format for executable files.

2.4.1 Windows

Windows is currently one of the most popular desktop operative systems, which is why a lot of malware target this platform. The Windows operative system is developed by Microsoft, and was originally based on their previous major operative system MS-DOS, although newer versions have mostly been rewritten without these parts. Windows uses the Portable Executable (PE) file format for its executable files [39]. This format was originally developed to cover all platforms, but is currently only used on the Windows operative system. It has remained more or less completely the same since its introduction in the 1980s, with the largest addition being support for 64-bit operative systems.

PE files are divided into a header and sections, as shown in figure 3. The header contains information about the file, while the sections contains code and other data used by the program during execution. The following sections describe the header and sections, as well as how code reuse is performed in the Windows operative system.

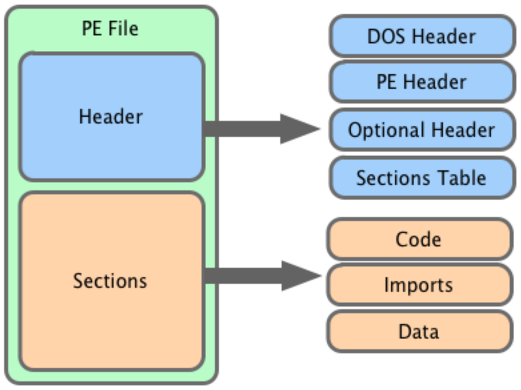


Figure 3: Structure of a PE file. From [2]

Header

The header contain different information required to load the file into memory and execute it, as well as other metadata such as author and compile time. It can be further divided into four parts:

- **DOS Header:** The first header is the DOS header, and is included for backwards compatibility. This header contains a small DOS program that is executed if one attempts to execute a PE file in such an environment. This small program usually prints the string "This program cannot be run in DOS mode", referred to as the MS-DOS stub, before exiting.

- **PE Header:** The second header is the PE header, and is located at hexadecimal offset *0x3C* in every PE file [40]. The PE header starts with the four byte signature $0x50, 0x45, 0x00, 0x00$, which, when interpreted as ASCII, reads as "PE" followed by two null-bytes. The rest of the header contains the architecture that the file is compiled for, the number of sections in the body of the file, the timestamp of when the field was compiled, and the size of the optional header, among some other fields. All fields are located at a constant offset from the start of the header, stored as either 16-bit WORDS or 32-bit DWORDS.
- **Optional Header:** Contradictory to what the name implies, this header is not optional, and is required in every PE file. Several of the fields provide necessary information to the operative system when the file is being loaded, such as the "AddressOfEntryPoint", which contains a pointer to the virtual address of the first instruction of the program. This header contains 60 fields where most of them can be set to arbitrary values without affecting execution. Some fields of interest, in addition to "AddressOfEntryPoint", are the "SizeOf..." and "BaseOf..." fields, which contains sizes and base addresses of the code and data segments, and the "ImageBase", which contains the base of the virtual address space in memory that the code will be mapped into.
- **Section table:** The section table lists the programs sections with metadata. This part is also required, as the operative system uses this information when loading the file. Each element

in the list represents a section in the file. The elements contains the name of the section, the size and location of the section in the file, and the size and virtual base address of the section when loaded into memory, among other information. The "Characteristics" filed contains flags that describe the section, for instance that a section contains code and is executable, or that it contains readable data.

Sections

The sections are stored within the main body of the PE file, as shown in figure 3. Table 1 contains a list of the most common section names, and what their used for. However, section names are completely arbitrary, and can be changed after compilation, as long as the corresponding element in the section table is also updated. The section characteristics on the other hand are not arbitrary. For instance, the ".text" section usually contains executable code, and must therefore have the "code", "readable" and "executable" flags set.

Table 1: Table of common PE sections with descriptions. From [6]

.text	Main program code- usually execute and read access only.
.data	Main initialized data code that is used by the program.
.rsrc	Contains the Windows Resource data.
.rdata	Read only data.
.reloc	Base relocations.
.debug	Debug information.
.idata	Imported function data.
.tls	Thread Local Storage. Data private to each thread.
.CRT	Data reserved for the 'C' Run-Time Libnrary.

The most common section types are:

- **Code:** Usually named *.text*. Contains the executable code of the program, usually set to read and execute only, to prevent programs changing their own code during execution.
- **Imports:** Usually named *.idata*. Contains information about imported functions that the program use. These functions are parts of shared libraries, and not part of the programs own code.
- **Data:** Usually named *.data*, or *.rdata* for read-only data. These sectiosn contain data used bu the program, such as strings or other values.

Code reuse

Code reuse is common in most operative systems, and on Windows it comes in the form of Dynamic—Link Libraries (DDL). This saves disc-space, and enables standardization of functionality. Windows supports both static lining, where library functions are included in to programs code, and dynamic linking, where library functions are stored in other files and imported during run-time. DLLs are PE files containing the same header and body as normal PE files, but also contains a list of exported functions that other programs can import and use. They normally don't have a entrypoint, and cannot be executed by themselves. It is possible to execute DLL files, but that usually involve using another program, like rundll32.exe, to import the DLL and call a specified function. A PE file is recognised as a DLL file by a particular flag in its PE header. More specifically, the "IMAGE_FILE_DLL" flag must be set in the "Characteristics" field. Setting this flag prevents execution of this file, but enables other programs to import its functions.

2.4.2 Linux

The Executable and Linkable Format (ELF) is the most common executable file format on most Linux based systems [9]. This format has several similarities to the PE format, but is less complex. The ELF format was introduced in the 1990s, and chosen as the standard executable file format for Unix-based systems in 1999. It is designed to be cross platform, and support extensions for later additional functionality.

ELF files starts with the byte sequence $0x7F, 0x45, 0x4C, 0x46$. The last 3 bytes in the sequence spells "ELF" when interpreted as ASCII. This signature, usually referred to as the magic number, is required for every ELF file.

An ELF file consists of the ELF header, followed by a program header table, section header table or both, and a body. The body contains sections of code and resources, which are refered to as segments when the file is mapped into memory. Figure 4 presents the layout of an ordinary ELF file, both on disk and in memory. We will base our description on the linking view of ELF files. The following sections describe the header and header tables, sections, and code reuse.

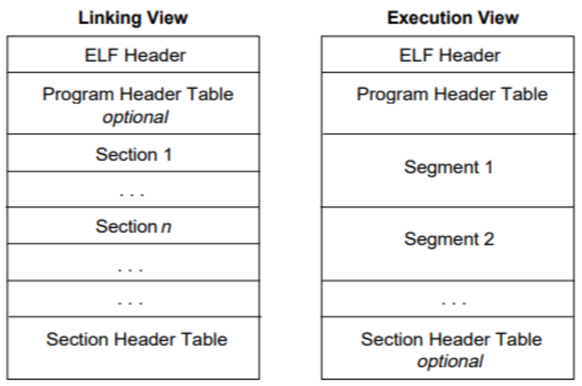


Figure 4: Executable and Linkable Format file layout. Linking view is the file on disk, while execution view is the file mapped into memory. From [3]

Headers

The headers contain information required to map the ELF file into memory and execute it. The header tables contain as entries the headers of sections or segments.

- **ELF Header:** The ELF header starts with the aforementioned magic number, and contains information required by the operative system when running the file. This includes whether it's compiled for 32-bit or 64-bit architecture, and which operative system the file supports. The 32-bit and 64-bit versions of the ELF header has the same fields, but some of the vary in size, as shown in listing 2.1. Additionally, the header contains the "e_entry" field, which points to the first instruction of the program.

- **Program Header Table:** The program header table is used to tell the operative system how to load the file into memory and prepare it for execution. Each element in the table describes a segment in the executable, and each segment will represent one or more section when the file is mapped into memory. The program header is required for most normal executables.
- **Section Header Table:** The section header list all sections in the file and metadata about them. The sections are used similarly the those of PE files. A list of common ELF sections are presented in table 2.

Listing 2.1: ELF header C struct. N is 32 or 64, such that ElfN becomes Elf32 or Elf64 etc. From [9]

```
#define EI_NIDENT 16

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    uint16_t      e_type;
    uint16_t      e_machine;
    uint32_t      e_version;
    ElfN_Addr     e_entry;
    ElfN_Off      e_phoff;
    ElfN_Off      e_shoff;
    uint32_t      e_flags;
    uint16_t      e_ehsize;
    uint16_t      e_phentsize;
    uint16_t      e_phnum;
    uint16_t      e_shentsize;
    uint16_t      e_shnum;
    uint16_t      e_shstrndx;
} ElfN_Ehdr;
```

Sections

The sections make up the main body of the ELF file on disk, and contains code and other resources used by the file during execution. The most common sections are presented in table 2. Each section is assigned one of three of access modes: read-only, read-write and read-execute. As with PE files, the section names are arbitrary, but they are usually the same when the file is compiled with normal compilers and not altered afterwards.

Code reuse

As on Windows, code reuse is common to save disk space and standardize common functionality, and ELF files can use shared code using both static and dynamic linking. Libraries that are linked to are in themselves ELF files. Dynamic linking is the most common method. One of the read-only sections, usually the ".dynamic" section, contains names of libraries and addresses of functions

Table 2: Table of common ELF sections with descriptions. From [3]

.bss	This section holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type, SHT_NOBITS.
.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.
.dynamic	This section holds dynamic linking information and has attributes such as SHF_ALLOC and SHF_WRITE. Whether the SHF_WRITE bit is set is determined by the operating system and processor.
.init	This section holds executable instructions that contribute to the process initialization code. When a program starts to run, the system executes 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.
.text	This section holds the "text," or executable instructions, of a program.

that the program links to and wishes to use. These functions are mapped into memory when the operative system loads the executable file. This is, however, less portable, as different operative systems and versions may store libraries at different locations, or use different versions of libraries that are not compatible.

Static linking fixes the problem of portability by including the linked functions when the program is compiled. This way, the right functions are always used, and the program can be executed on any system using the same architecture. The downsides of this method is that the ELF files become much larger, as all library code must be included, and that it becomes much more difficult to update the library code without having to get a new version of the program built using the new library. Other executable file types

2.5 Malware analysis

Malware analysis is a collective term for techniques and methods used to extract information about and understand behavior and capabilities of malware. The goal of malware analysis is usually to provide some information about a given malware sample to incident response personnel, to help determine the effects of an incident. Malware analysis may also be used to create signatures for use in detection (described in section 2.7), to classify malware based on some metric (described in section 2.8), and to uncover information regarding the origin of the malware sample and the origination behind it. Malware analysis techniques are usually divided into two groups: static and dynamic. In short, static techniques involves analyzing the malware without executing it, while dynamic techniques involve running it and observing its behavior. Additionally, Sikorski and Honig further divides these two groups in two [11], by distinguishing between basic and advanced techniques. These four groups are summarized below.

Basic static analysis
Examine header and metadata without execution.
Pros: Quick and easy. Can get some basic signatures.
Cons: Insufficient with sophisticated malware. Offers little information.

Advanced static analysis
Reverse engineering and analyzing instructions, without execution. Often by disassembling the malware and reading assembly code.
Pros: Tells you exactly what the malware does. Can make more detailed signatures.
Cons: Slow and difficult. Requires a lot of knowledge.

Basic dynamic analysis
Execute malware and observe changes to system. Usually in a sandbox with simulated internet and filesystem.
Pros: Easy. See what the malware does e.g. to the filesystem, registry or network.
Cons: Requires secure environment. Fails with more complex malware.

Advanced dynamic analysis
Use debugger to observe state of malware. Record changes to one or several systems.
Pros: More details about what the malware does. Can circumvent anti-analysis techniques
Cons: Slow and difficult. Requires a lot of setup.

2.6 Automatic malware analysis

Automatic malware analysis employ static and dynamic analysis techniques to recover information about a given malware sample without requiring human interaction. This saves time, and gives a human analyst initial information when starting analysis of a new sample. In its simplest form, automatic analysis employ techniques from basic static analysis. These simply read or generate metadata about the malware file, and can easily be automated. This speeds up the analysis of new malware, and may give incident responders required information quickly, but may easily be fooled by more advanced malware.

To improve automatic analysis, several vendors add techniques from basic dynamic analysis. Usually, this comes in the form of sandboxes. A sandbox is closed, secure, and prepared environment where the malware can be executed and recorded. These are usually based on virtual machines, and are automatically created, recorded and cleaned up when a new malware sample is tested. The sandbox records changes to and actions on the computer. This may include changes to files or registry, or network traffic generated by the malware. After execution, a report is generated that summarizes all actions taken by the malware sample. This report can give the malware analyst or incident response team valuable information towards understanding an incident.

One major problem of sandboxes is to create the correct environment for the malware sample. Performing basic static analysis beforehand may aid in this, to determine which environment to create for the new sample. This, however, may only get so far, as some malware may include anti-analysis techniques or anti-virtual machine techniques. By combining basic static and dynamic techniques, most normal malware can be analyzed, but the method is still insufficient when analyzing more advanced malware.

Employing advanced techniques automatically, both static and dynamic, is less common. These techniques usually require an individual to constantly interpret observations and make decisions as to what actions to take. However, they can to some degree be used, as long as a human analyst is used afterwards to evaluate the output. Debuggers, from advanced dynamic analysis, may be used during execution to record the internal state of the sample at given intervals or when particular events occur. For instance, all function calls can be recorded and constructed into a directed graph, as was done by Østbye in [41]. This provides additional information, but does not utilize the full potential of advanced dynamic analysis. For instance, stopping execution at interesting internal states still requires a human analyst to first determine what constitutes an internal state.

Techniques from advanced static analysis may also be employed, but suffers from the same drawbacks as the once from advanced dynamic. Assembly instructions and other information can automatically be recovered from a malware sample, but required either a human or some other mechanism to determine which parts are interesting and what actions to take.

2.7 Malware detection

Malware detection is the process of analyzing files to determine if a given file contains malicious code or not. This is an example of a binomial classification problem, where the two classes are benign and malware. Often, this involves employing automatic analysis, combined with some metric to determine the class of a file. Anti-virus products usually employ some form of malware detection to protect computers.

Static signature-based detection is the most primitive form of malware detection. Known malware are analysed, and byte-sequences that are more or less unique to those samples are recovered. These byte-sequences are referred to as signatures, and can be used for detection by finding files that contains strings that match. If a match is found in a file, then the file is probably the same as the analysed malware, depending on how unique the signature is. These signatures can further be improved by including wildcards, that can represent from zero to many bytes of unspecified value, or implementing a full regular expression engine [42]. This enable detection of new versions of malware with small changes, and allows circumvention of simple obfuscation techniques. However, all static signatures suffer from the drawback that new malware samples must first be analysed to create signatures, and can therefore only detect known malware or known malware with minor modifications.

To detect new malware, we must add a layer of abstraction to the process. Instead of detection malicious files, we can attempt to detect malicious behaviour. Behaviour-based signatures intend to capture some behaviour considered to be malicious. This is most easily based on dynamic analysis techniques, where the behaviour is recorded, but can also be used with static analysis, that in some way captures and represent behaviour based on code analysis. By using behaviour-based signatures, as opposed to byte-sequences, we are able to detect all malware that conducts the same activities. Developing good behaviour signatures, however, is not easy, as determining which of the several activates performed by a file should be considered malicious is a difficult task.

2.8 Malware classification

Determining that a file is malicious is useful information to an incident responder, but additional information can be provided by determining to which previously observed malware samples the new sample is most similar. Malware classification is not one, but several multinomial classification problems, as there exists several ways to group malware. As with detection, malware classification employ some form of automatic analysis with one or more metrics. Where classification differs from detection, is that the metrics must support calculation of similarity, and not only support exact matching. In general, malware classification groups malware in some way, and uses one or more metrics derived from analysis to measure similarity between a new sample and the different groups. The new sample is placed in to groups with the highest similarity.

As an example, malware classification can be performed using static byte-sequence signatures and inexact matching to measure similarity between a new malware sample and existing signatures representing the different groups. The malware then belongs to the group represented by the signature with the highest similarity. Additionally, multiple signatures, both static and behavioural, can be used to represent a single group, and the similarity score towards a group is determined by the (possibly weighted) sum of similarity scores towards the signatures of that group. This enables classification of new malware within the existing classes, but may fail if a new malware sample does not belong to one of the groups. Some vendors have solved this by not assigning samples to a group if none of the similarity scores are above a certain threshold. This creates an implicit, extra group of “unclassified” samples. In some cases, these samples may be the most interesting, as they may be completely new.

2.9 Malware class sets

As stated in the previous section, malware classification is more than one multinomial classification problem. Before classifying malware, a class-set must be determined. This is the set of possible classes that may be assign to samples during classification. Each class in the set must describe the same information about different samples, and each sample should ideally only belong to one class. For instance, if we were to classify cars, we can choose a class-set consisting of different brands (Ford, Audi, Toyota, etc.), or we can choose one consisting of different fuel-types (petrol, diesel, electric, etc.). However, if we created a class-set that included some elements from both these sets, the resulting set would not make any sense, as the different classes represent different information and a given car may belong to multiple classes.

In sections 2.2 and 2.3, we describe two different ways to group malware: type and family. In this section, we add a third class-set: attribution. These three are the most common class-sets used when classifying malware, and cover what the malware does (type), how it does it (family), and who made it (attribution). There may exist other class-sets for malware classificationt, but these are not included in this thesis.

Malware type: the general functionality of malware. Section 2.2 presents a list of common types, and these may be used to create a class-set [43]. However, classifying malware by types is difficult, as the type only describes what the malware does. Two viruses, for instance, may be designed to infect files of different types. Their code may therefore be completely different, and we must therefore capture their behaviour when performing classification. Additionally, as described in section 2.2.11, malware may contain functionality from multiple types.

Malware family: the specific functionality, and how they achieve it [43]. Section 2.3 describes malware families in more detail. Malware belonging to the same family has more in common than those belonging to the same type, especially with regards to code and code structure. There does however exist a very large number of different families, which makes creating a class-set of all families almost impossible. As a result, a "unknown" class may be required when this class-set is used in real life scenarios, to capture any samples that does not belong in any of the known families.

Attribution: classifying malware by its author or origin. This class-set attempts to group malware by their origin, by finding similarities in, for instance, coding styles or obfuscation techniques. Being able to correctly attribute an attack may provide valuable information to the incident response team, to determine the intent of the attacker, and may be used after the attack when taking legal actions. If a previous malware samples has been attributed to a specific threat actor, new samples may be attributed to the same actor if they are in some way similar to the original. This form of classification was performed in both [44] and [45], although they used benign software instead of malware. This class-set sufferers from the same drawback as malware families, as there exist an unknown number of malware authors. Additionally, obtaining a large number of attributed malware samples, to study and train a classifier with, may prove difficult. With malware families, we can obtain different versions of one sample, but with attribution, we may also need different malware families created by the same author, so that we do not simply classify families while using attribution as class labels.

2.10 Anti-analysis techniques

Many developers, and especially malware developers, do not wish to have their software analyzed or reverse engineered. Developers of benign software may wish to hide algorithms or design details that may be stolen by competing vendors, while malware developers may wish to hinder or slow down analysis, or to avoid detection altogether [46]. Anti-analysis techniques cover any changes or additions that developers add to their software that do not change its functionality, but increases the work required to analyze the software. Time is often limited when handling an incident, and analyzing malware may provide essential information if done quickly. All malware can eventually be analyzed, but slowing down this process can increases the likelihood of success for the attacker.

Different anti-analysis techniques target different types of analysis. In the rest of this section, we describe some common anti-analysis techniques that are relevant to this thesis, grouped by which type of analysis they target.

2.10.1 Techniques against static analysis

Static analysis techniques retrieve information about a malware sample without executing it, by examining metadata and code. Most anti-analysis techniques targeting static analysis use some form of code obfuscation, in an attempt to hide what the code does. Changes are made to the code, before or after compilation, to mask its intent, increase the amount of work required to interpret the code, or alter the code signature, without changing the behaviour of the code. As testing the effects of obfuscation is not the focus of this thesis, we have chosen to limit the presentation to those techniques that are used after compilation.

This section lists several common obfuscation techniques targeting static analysis. The first four are simple techniques, and are also presented in figure 5. The last three are more advanced techniques that usually affect a larger portion of the program.

Register reassignment: change which registers values are stored in. In figure 5 (b), registers eax, ebx and edx in the original code (a) have been reassigned to ebx, edx and eax, respectively.

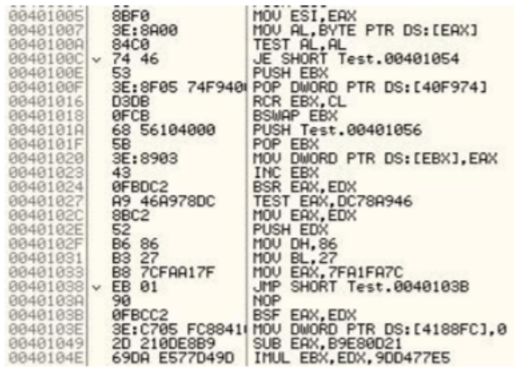
Code substitution: change on or more instructions to different once that produce the same result. In figure 5 (c), the red boxes highlights two instructions that have be changed from "test" instructions in the original code (a), to "or" instructions in (c).

Dead code insertion: add additional code to the malware in between existing instructions that has no effect on the behaviour. In figure 5 (d), the red box highlights a sequence of code that has no additional effect on the programs behaviour. The sequence executes the "nop" instruction twice, which has no effect, increments the value of ebx, pushes the value of ebx onto the stack (as it does in the original code), before decrementing the value ontop of the stack and the value of edx. The state of the registers and stack is therefore the same at next instruction after the box in (d) as it was at the same instruction in the original code (a).

Code transposition: reorder code by adding or removing unnecessary jumps between instructions. In figure 5 (e), the colored boxes represents the sequences of code from the original (a) that have been reordered, and the arrows show the new order. The order of execution remains the same, but the order in memory has been changed by dividing the instructions into random, non-overlapping sequences, reordering them, and inserting jumps to the next sequence at the end of each one.

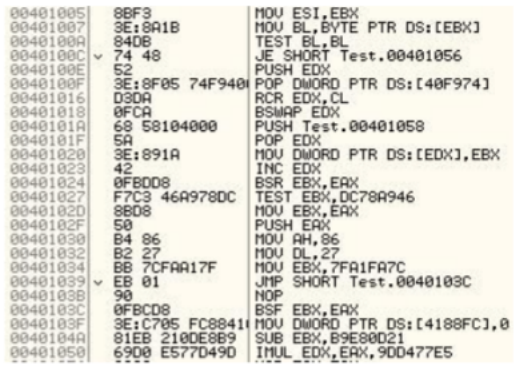
Innlining and outlining: innlining is the process of removing function calls by inserting the functions code into the callers code where the original call was made. Outlining is the opposite, where random bits of code is removed, made into a function, and a call to the function is inserted where the original code was. Both of these alters the control flow, while producing the same result [47][48].

Indirect jumps: target address of a jump is determined at runtime, usually by jumping to a value stored in a register [48]. These are often necessary, for instance in jump tables, but can also be used to obfuscate the target of a jump [49]. A normal jump can be converted to an indirect jump by first storing the jump target in a registry, and jumping to the value of the registry. Simple techniques like this are deobfuscated by a human analyst, but can often fool automatic analysis.



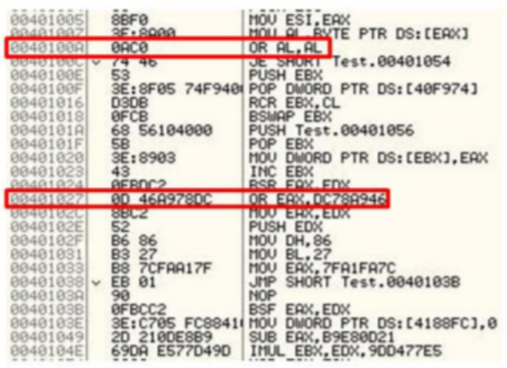
```
00401005   8BF0              MOV ESI,EAX
00401007   3E:8A00           MOV AL,BYTE PTR DS:[EAX]
0040100A   84C0              TEST AL,AL
0040100C v 74 46             JE SHORT Test.00401054
0040100E   53                PUSH EBX
0040100F   3E:8F05 74F940(   POP DWORD PTR DS:[40F974]
00401016   D3DB              RCR EBX,CL
00401018   0FCB              BSWAP EBX
0040101A   68 56104000       PUSH Test.00401056
0040101F   5B                POP EBX
00401020   3E:8903           MOV DWORD PTR DS:[EBX],EAX
00401023   43                INC EBX
00401024   0FBDC2            BSR EAX,EDX
00401027   A9 46A978DC       TEST EAX,DC78A946
0040102C   8BC2              MOV EAX,EDX
0040102E   52                PUSH EDX
0040102F   B6 86             MOV DH,86
00401031   B3 27             MOV BL,27
00401033   B8 7CFAA17F       MOV EAX,7FA1FA7C
00401038 v EB 01             JMP SHORT Test.0040103B
0040103A   90                NOP
0040103B   0FBCC2            BSF EAX,EDX
0040103E   3E:C705 FC8841(   MOV DWORD PTR DS:[4188FC],0
00401049   2D 210DE8B9       SUB EAX,B9E80D21
0040104E   69DA E577D49D     IMUL EBX,EDX,9DD477E5
```

(a) Sample code



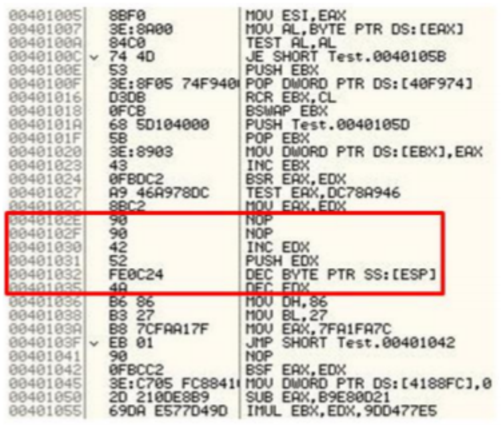
```
00401005   8BF3              MOV ESI,EBX
00401007   3E:8A1B           MOV BL,BYTE PTR DS:[EBX]
0040100A   84DB              TEST BL,BL
0040100C v 74 48             JE SHORT Test.00401056
0040100E   52                PUSH EDX
0040100F   3E:8F05 74F940(   POP DWORD PTR DS:[40F974]
00401016   D3DA              RCR EDX,CL
00401018   0FCA              BSWAP EDX
0040101A   68 58104000       PUSH Test.00401058
0040101F   5A                POP EDX
00401020   3E:891A           MOV DWORD PTR DS:[EDX],EBX
00401023   42                INC EDX
00401024   0FBDD8            BSR EBX,EAX
00401027   F7C3 46A978DC     TEST EBX,DC78A946
0040102D   8BD8              MOV EBX,EAX
0040102F   50                PUSH EAX
00401030   B4 86             MOV AH,86
00401032   B2 27             MOV DL,27
00401034   BB 7CFAA17F       MOV EBX,7FA1FA7C
00401039 v EB 01             JMP SHORT Test.0040103C
0040103B   90                NOP
0040103C   0FBCD8            BSF EBX,EAX
0040103F   3E:C705 FC8841(   MOV DWORD PTR DS:[4188FC],0
0040104A   81EB 210DE8B9     SUB EBX,B9E80D21
00401050   69D0 E577D49D     IMUL EDX,EAX,9DD477E5
```

(b) Register reassignment



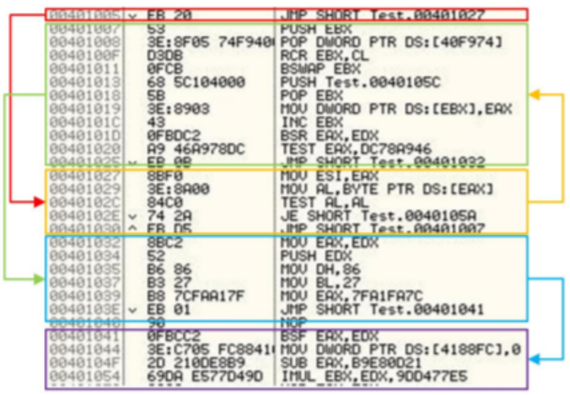
```
00401005   8BF0              MOV ESI,EAX
00401007   3E:8A00           MOV AL,BYTE PTR DS:[EAX]
0040100A   0AC0              OR AL,AL
0040100C v 74 46             JE SHORT Test.00401054
0040100E   53                PUSH EBX
0040100F   3E:8F05 74F940(   POP DWORD PTR DS:[40F974]
00401016   D3DB              RCR EBX,CL
00401018   0FCB              BSWAP EBX
0040101A   68 56104000       PUSH Test.00401056
0040101F   5B                POP EBX
00401020   3E:8903           MOV DWORD PTR DS:[EBX],EAX
00401023   43                INC EBX
00401024   0FBDC2            BSR EAX,EDX
00401027   0D 46A978DC       OR EAX,DC78A946
0040102C   8BC2              MOV EAX,EDX
0040102E   52                PUSH EDX
0040102F   B6 86             MOV DH,86
00401031   B3 27             MOV BL,27
00401033   B8 7CFAA17F       MOV EAX,7FA1FA7C
00401038 v EB 01             JMP SHORT Test.0040103B
0040103A   90                NOP
0040103B   0FBCC2            BSF EAX,EDX
0040103E   3E:C705 FC8841(   MOV DWORD PTR DS:[4188FC],0
00401049   2D 210DE8B9       SUB EAX,B9E80D21
0040104E   69DA E577D49D     IMUL EBX,EDX,9DD477E5
```

(c) Code substitution



```
00401005   8BF0              MOV ESI,EAX
00401007   3E:8A00           MOV AL,BYTE PTR DS:[EAX]
0040100A   84C0              TEST AL,AL
0040100C v 74 4D             JE SHORT Test.0040105B
0040100E   53                PUSH EBX
0040100F   3E:8F05 74F940(   POP DWORD PTR DS:[40F974]
00401016   D3DB              RCR EBX,CL
00401018   0FCB              BSWAP EBX
0040101A   68 5D104000       PUSH Test.0040105D
0040101F   5B                POP EBX
00401020   3E:8903           MOV DWORD PTR DS:[EBX],EAX
00401023   43                INC EBX
00401024   0FBDC2            BSR EAX,EDX
00401027   A9 46A978DC       TEST EAX,DC78A946
0040102C   8BC2              MOV EAX,EDX
0040102E   90                NOP
0040102F   90                NOP
00401030   42                INC EDX
00401031   52                PUSH EDX
00401032   FE0C24            DEC BYTE PTR SS:[ESP]
00401035   4A                DEC EDX
00401036   B6 86             MOV DH,86
00401038   B3 27             MOV BL,27
0040103A   B8 7CFAA17F       MOV EAX,7FA1FA7C
0040103F v EB 01             JMP SHORT Test.00401042
00401041   90                NOP
00401042   0FBCC2            BSF EAX,EDX
00401045   3E:C705 FC8841(   MOV DWORD PTR DS:[4188FC],0
00401050   2D 210DE8B9       SUB EAX,B9E80D21
00401055   69DA E577D49D     IMUL EBX,EDX,9DD477E5
```

(d) Dead code insertion



```
00401005 v EB 20             JMP SHORT Test.00401027
00401007   53                PUSH EBX
00401008   3E:8F05 74F940(   POP DWORD PTR DS:[40F974]
0040100F   D3DB              RCR EBX,CL
00401011   0FCB              BSWAP EBX
00401013   68 5C104000       PUSH Test.0040105C
00401018   5B                POP EBX
00401019   3E:8903           MOV DWORD PTR DS:[EBX],EAX
0040101C   43                INC EBX
0040101D   0FBDC2            BSR EAX,EDX
00401020   A9 46A978DC       TEST EAX,DC78A946
00401025 v EB 0B             JMP SHORT Test.00401032
00401027   8BF0              MOV ESI,EAX
00401029   3E:8A00           MOV AL,BYTE PTR DS:[EAX]
0040102C   84C0              TEST AL,AL
0040102E v 74 2A             JE SHORT Test.0040105A
00401030 ^ EB D5             JMP SHORT Test.00401007
00401032   8BC2              MOV EAX,EDX
00401034   52                PUSH EDX
00401035   B6 86             MOV DH,86
00401037   B3 27             MOV BL,27
00401039   B8 7CFAA17F       MOV EAX,7FA1FA7C
0040103E v EB 01             JMP SHORT Test.00401041
00401040   90                NOP
00401041   0FBCC2            BSF EAX,EDX
00401044   3E:C705 FC8841(   MOV DWORD PTR DS:[4188FC],0
0040104F   2D 210DE8B9       SUB EAX,B9E80D21
00401054   69DA E577D49D     IMUL EBX,EDX,9DD477E5
```

(e) Code transposition

Figure 5: Presentation of common obfuscation techniques targeting static analysis. (a) is the original sample code, and (b), (c), (d) and (e) show how the different obfuscation techniques, register reassignment, code substitution, dead code insertion and code transposition, respectively, alter (a) . From [4]

Packing: the code of the malware is compressed (and often also encrypted), and wrapped in another executable [11]. The other executable contains a decryption and decompression function that unpacks the original malware, maps it into executable memory, and executes it. Tools, called packers, have been created to pack existing executables, making this technique almost trivial to employ. Packing prevents static analysis, as the malware cannot be analyzed without unpacking it first. This can be done by running the decryption function, and writing the malware to file instead of executing it. For common packers, there exist tools, unpackers, that does this automatically. However, if a custom packer is used, the process must be done manually.

2.10.2 Techniques against dynamic analysis

Dynamic analysis involves executing the malware and record or observe changes to the computer or network. To counter dynamic analysis, malware developers have implemented checks in their malware that can determine if the malware is being dynamically analyzed. If the malware determines that it is being analyzed or executed in an unexpected, secure environment, it stop itself to not reveal its intended actions. Some samples also go as far as attempting to delete the executable binary on disk when they detect a debugger or unexpected environment before terminating, to prevent analysis [46].

Techniques targeting dynamic analysis can be divided into two categories: debugger detection (or anti-debugging) and environment detection (or anti-virtual machine) [50][51]. They focus on detecting analysis in different ways, both with the goal of changing their behaviour if analysis is detected. As this thesis primarily focuses on static analysis, we will only give a brief overview of these techniques. For further detail, see [11].

2.10.3 Debugger detection

Debugger detection is often referred to as anti-debugging, these techniques attempt to detect if the malware is being debugged during execution. Debuggers can inspect the internal state of a program, and manipulate it during execution by modifying code or values in registries or memory. Malware analysts often use debuggers to determine the functionality malware samples, as the sample may not present all its functionality during normal execution.

Some known techniques for debugger detection are:

Windows api: the Windows api [52] offers functionality, such as debugging events and functions, that enables a program to determine if its being debugged [53]. These are functions that the program can import and call, and the information they return directly or indirectly indicates that a debugger is present. These can easily be tricked, however, by using the debugger to interrupt the program before one of these functions return, and changing the return value.

Timing attacks: usually when a program is run under a debugger, the analyst steps through parts of it manually. If the program can measure how long a certain function took to execute, it can compare the result to a predetermined baseline, and determine whether or not it is being debugged. Timing attacks are easy to implement, as there are Windows api functions (e.g. GetTickCount)[52] and x86 assembly instructions (e.g. rdtsc)[54] for getting the current timestamp.

Code checksums: malware can use checksums to determine if its code has been altered before or during runtime [55]. Before executing a particular function, the malware can compute the checksum of the code within the function, and compare the value to a pre-calculated value.

2.10.4 Environment detection

Environment detection involve attempting to determine the environment that the malware is being executed in. If the malware is able to determine that it is being executed in a secure environment, it can change its functionality or terminate. The opposite can also be done, where the malware will only continue execution if it detects one particular computer or a small set of computers.

Usually, dynamic analysis is conducted in virtual machines. Therefore, a lot of malware focus on detecting artifacts of virtual machines. These artifacts are files or other configurations that are typical to virtual machines, and uncommon in normal machines. Some examples are "VMware-Service.exe", used for the virtual machine host when interacting with the virtual machine, and MAC-addresses of virtual network cards, that are set by default using specific prefixes [56].

2.11 Control Flow Graphs

This section presents an introduction to the concept of control flow graphs, and the ways they can be recovered form executable programs. Following this, we present some methods for comparing graphs and computing similarity scores, which can be used to classify graphs.

2.11.1 Introduction to control flow graphs

Analyzing a program by modelling its control flow as a graph is not a new technique [57][58]. A control flow graph is a way to model the execution flow of a program as a directed graph. This technique aim to recover a programs instructions in the order they are executed, while maintaining the possible execution sequences that may occur based on conditional branches.

A programs executable code can be divided into a set of blocks, called basic blocks [59], that each represents a finite string of instructions between two branching instructions. This representation simplifies analysis, as a string of instructions in a basic block will always be executed in the order they are presented. The first block in the set starts at the programs entry-point. Every block after this starts at the target of a previously executed branch instruction. A block ends either when it encounters a branching instruction, the final instruction of the program, or an instruction that is the target of a branching instruction. The nodes in a control flow graph represent these basic blocks. Nodes are often labelled with the start address of the block they represents, and may contain attributes that store other relevant information about the block, such as its sequence of instructions.

The edges in a control flow graph represent the flow of control between the basic blocks. The control flow graph is a directed graph, as branching instructions are only single direction. Edges directed towards a node represent the possible jumps that targets the first instruction of the basic block represented by the node, while edges directed away from a node represent branches to the start of other blocks. A node may have an unlimited number of incoming edges, but only zero, one or two outgoing edges. If a node has zero outgoing edges, the program exits after that node.

One outgoing edge indicates a normal control flow transfer, while two outgoing edges indicate conditional branching, where a branch is chosen during execution based on some condition. As with the nodes, edges may also contain additional information using labels or attributes. An example of this is labelling the edges with the type of branching instruction they represent.

2.11.2 Definitions

The control flow graph is a directed graph where the set of nodes represent basic blocks, and the set of edges represent control flow between the blocks. We use these terms in this thesis, and have therefore included formal definitions for completeness.

Definition 1 (*Graph*): A graph *G* consists of a set of nodes *V(G)* and a set of edges *E(G)* representing the relationships between the nodes. That is, $\forall(u,v) \in E(G)$ if and only if $(u,v) \in V(G)$. [59]

Definition 2 (*Directed graph*): Graph *G* is considered to be a directed graph if edges *E(G)* have a direction associated with them. That is, for nodes $\forall(u,v) \in V(G)$, edges $((u,v),(v,u)) \in E(G)$ are not considered identical. The opposite is an un-directed graph, where *(u, v)* and *(v, u)* are identical, and *(u, v)* implies *(v, u)*. [60]

Definition 3 (*Sub-graph*): A graph *H* is defined as a sub-graph of a graph *G* if and only if $|V(H)| \leq |V(G)|$ and there exists and injective function $f : V(H) \mapsto V(G)$, such that $\forall(u,v) \in E(H)$ the relation $(f(u), f(v)) \in E(G)$ holds. [60]

Definition 4 (*Graph isomorphism*): Graphs *G* and *H* are isomorphic if and only if $|V(H)| = |V(G)|$ and there exists a bijective function $f : V(H) \mapsto V(G)$, such that $\forall(u,v) \in E(H)$ the relation $(f(u), f(v)) \in E(G)$ holds. That is, both graphs are a sub-graph of the other. [60]

It is also possible to assign additional information to nodes and edges through labels and attributes. A label is a symbol that describes the node or edge that it is assigned to. Two or more elements in a graph may have the same label, while still being considered unique elements, as their position in the graph may still offer some information. Labels are usually strings, numbers, colors or some other symbol. Assigning labels to the elements of a graph increases the amount of information that can be represented within the graph and helps distinguish between different sub-graphs of a graph. The labels can be used as unique identifiers for each node/edge or divide them into categories. What labels are selected for the nodes and edges are highly dependent on the classification method used.

In addition to labels, it is also possible to assign information to nodes and edges using attributes. An attribute is usually represented by a key-value pair. Usually, every node in a graph have the same set of attribute keys with varying values. The same is true for the set of edges. As with the labels, the attributes add additional information to the graph, and some or all of them may be considered when analyzing and comparing different graphs or sub-graphs.

2.11.3 Different representations

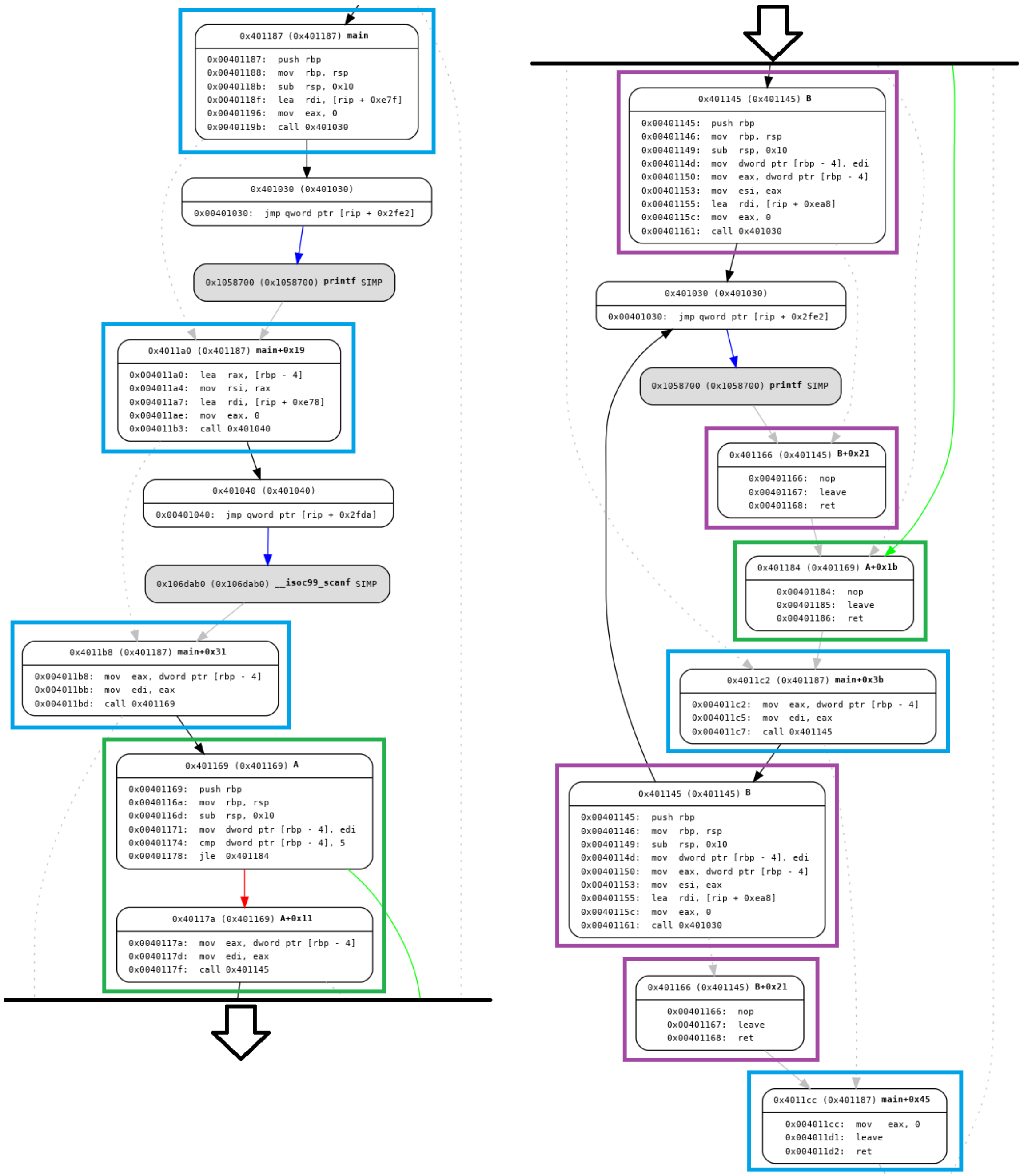
Two different ways of representing a control flow graph has been identified: single graph and multiple graphs. In the first representation, the whole program and all its basic blocks are included in a single, large graph. Function calls are considered to be branching instructions, and will therefore end blocks the same way other branching instructions does. [61] uses this representation when recovering control flow graphs.

In the second representation, an individual graph is recovered for every function in the program. Some analysis must be conducted beforehand to determine the start and end of every function on the program. The control flow graph of a program will then consist of a set of smaller, unlinked graphs. Function calls are in this representation not considered to be branching instructions. It is possible to determine how graphs are linked by finding function calls within a function and checking which function is being called. [62] represents control flow graphs in this way.

An example of the different representations is presented in figures 6 for single graph (recovered with angr [61]) and figure 7 for multiple graphs (recovered with IDA Pro [62]). These graphs are recovered from the dummy program presented in listing 2.2. The dummy program, simply asks the user to input a number and calls functions A and B with the number as an argument. Function A checks if the number is greater than five, and calls B if the condition is true, to highlight how conditional branching and nesting of function calls are shown in the different graph representations. Function B simply prints the number, and is used to show how calling the a function at different locations in the code is represented.

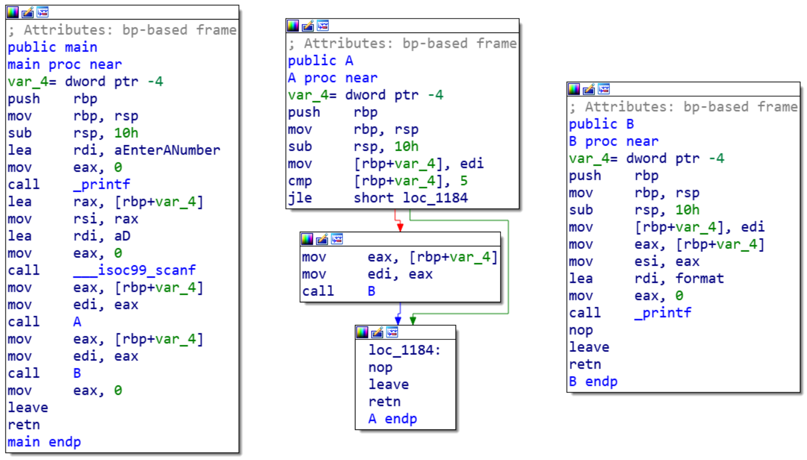
Listing 2.2: Dummy C program to show differences in control flow graph representations. The main function asks for a number, and calls functions A, which checks the number and calls B, which prints the number

```c
#include <stdio.h>

void B(int number){
    printf("You␣entered:␣%d\n", number);
}

void A(int number){
    if (number > 5){
        B(number);
    }
}

int main(){
    int number;

    printf("Enter␣a␣number:␣");
    scanf("%d", &number);

    A(number);
    B(number);

    return 0;
}
```



```
0x401187 (0x401187) main
0x00401187:  push rbp
0x00401188:  mov  rbp, rsp
0x0040118b:  sub  rsp, 0x10
0x0040118f:  lea  rdi, [rip + 0xe7f]
0x00401196:  mov  eax, 0
0x0040119b:  call 0x401030

0x401030 (0x401030)
0x00401030:  jmp qword ptr [rip + 0x2fe2]

0x1058700 (0x1058700) printf SIMP

0x4011a0 (0x401187) main+0x19
0x004011a0:  lea  rax, [rbp - 4]
0x004011a4:  mov  rsi, rax
0x004011a7:  lea  rdi, [rip + 0xe78]
0x004011ae:  mov  eax, 0
0x004011b3:  call 0x401040

0x401040 (0x401040)
0x00401040:  jmp qword ptr [rip + 0x2fda]

0x106dab0 (0x106dab0) __isoc99_scanf SIMP

0x4011b8 (0x401187) main+0x31
0x004011b8:  mov  eax, dword ptr [rbp - 4]
0x004011bb:  mov  edi, eax
0x004011bd:  call 0x401169

0x401169 (0x401169) A
0x00401169:  push rbp
0x0040116a:  mov  rbp, rsp
0x0040116d:  sub  rsp, 0x10
0x00401171:  mov  dword ptr [rbp - 4], edi
0x00401174:  cmp  dword ptr [rbp - 4], 5
0x00401178:  jle  0x401184

0x40117a (0x401169) A+0x11
0x0040117a:  mov  eax, dword ptr [rbp - 4]
0x0040117d:  mov  edi, eax
0x0040117f:  call 0x401145

0x401145 (0x401145) B
0x00401145:  push rbp
0x00401146:  mov  rbp, rsp
0x00401149:  sub  rsp, 0x10
0x0040114d:  mov  dword ptr [rbp - 4], edi
0x00401150:  mov  eax, dword ptr [rbp - 4]
0x00401153:  mov  esi, eax
0x00401155:  lea  rdi, [rip + 0xea8]
0x0040115c:  mov  eax, 0
0x00401161:  call 0x401030

0x401030 (0x401030)
0x00401030:  jmp qword ptr [rip + 0x2fe2]

0x1058700 (0x1058700) printf SIMP

0x401166 (0x401145) B+0x21
0x00401166:  nop
0x00401167:  leave
0x00401168:  ret

0x401184 (0x401169) A+0x1b
0x00401184:  nop
0x00401185:  leave
0x00401186:  ret

0x4011c2 (0x401187) main+0x3b
0x004011c2:  mov  eax, dword ptr [rbp - 4]
0x004011c5:  mov  edi, eax
0x004011c7:  call 0x401145

0x401145 (0x401145) B
0x00401145:  push rbp
0x00401146:  mov  rbp, rsp
0x00401149:  sub  rsp, 0x10
0x0040114d:  mov  dword ptr [rbp - 4], edi
0x00401150:  mov  eax, dword ptr [rbp - 4]
0x00401153:  mov  esi, eax
0x00401155:  lea  rdi, [rip + 0xea8]
0x0040115c:  mov  eax, 0
0x00401161:  call 0x401030

0x401166 (0x401145) B+0x21
0x00401166:  nop
0x00401167:  leave
0x00401168:  ret

0x4011cc (0x401187) main+0x45
0x004011cc:  mov  eax, 0
0x004011d1:  leave
0x004011d2:  ret
```

Figure 6: Control flow graph of the dummy program in listing 2.2, using single graph representation. Blue nodes are part of the main function, green of function A, and purple of function B. The control flow, represented by the arrows, jumps between nodes of different functions, as they are all part of the same graph. The unmarked nodes represents parts of library functions. The rest of the library functions have been removed for readability.



```
; Attributes: bp-based frame
public main
main proc near
var_4= dword ptr -4
push    rbp
mov     rbp, rsp
sub     rsp, 10h
lea     rdi, aEnterANumber
mov     eax, 0
call    _printf
lea     rax, [rbp+var_4]
mov     rsi, rax
lea     rdi, aD
mov     eax, 0
call    ___isoc99_scanf
mov     eax, [rbp+var_4]
mov     edi, eax
call    A
mov     eax, [rbp+var_4]
mov     edi, eax
call    B
mov     eax, 0
leave
retn
main endp
```

(a) Function main

```
; Attributes: bp-based frame
public A
A proc near
var_4= dword ptr -4
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], edi
cmp     [rbp+var_4], 5
jle     short loc_1184
```

```
mov     eax, [rbp+var_4]
mov     edi, eax
call    B
```

```
loc_1184:
nop
leave
retn
A endp
```

(b) Function A

```
; Attributes: bp-based frame
public B
B proc near
var_4= dword ptr -4
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], edi
mov     eax, [rbp+var_4]
mov     esi, eax
lea     rdi, format
mov     eax, 0
call    _printf
nop
leave
retn
B endp
```

(c) Function B

Figure 7: Control flow graph of the dummy program in listing 2.2, using multiple graphs representation. (a) is a single node, representing the main function. (b) and (c) represent functions A and B, respectively. Function A is split into three nodes, due to the conditional branching.

2.11.4 Control flow graph recovery

Recovering the control flow graph of a program relies on finding every basic block with the program. This is done by finding every branching instruction within the program, as well as every instruction that is the target of a branching instruction. This can be done using both static and dynamic analysis methods. A naïve method for control flow graph recovery using dynamic analysis is simply executing the program under a debugger, and recording the address of every branching instruction executed, as well as its target. This method is excellent at recovering jumps and their targets, but has limited coverage as it will only recover the jumps and basic blocks that have been executed. More advanced methods include forced execution, where the state of the program is altered to force different execution paths, combined with program state recovery and exception handling, which stores the programs state at branching instructions such that different paths may be explored [63].

In this thesis, we focus on the recovery methods that rely on static analysis. Primarily, these methods rely on reading the programs executable binary, and parsing the instructions. After determining sections and entry points, it is possible to follow the instructions and the control flow. From this, the set of basic blocks and branches of the programs can be recovered. Allen presents in [57] several procedures that can be used to recover the control flow graph of a program. These procedures are old, but are still referenced in more modern literature [64][65]. A more modern method was presented by Kinder in [66], where the instructions were translated into a intermediate language that breaks down complex assembly instructions into sequences of statements that represent the semantics of the machine code. The intermediate representation is the used to build the control flow graph. Other methods are more similar to the method presented by Allen, using recursive algorithms similar to the depth first search algorithm to iterate over and recover every basic block in the program [61].

2.11.5 Difficulties

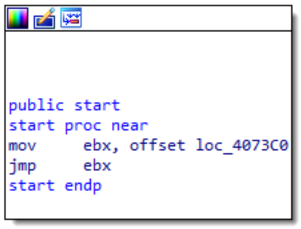
The primary challenge of using static analysis to recover the control flow graph is determining the target of indirect jumps. With direct jumps, the target of the jump is encoded into the instruction, and is easy to recover. Indirect jumps on the other hand, reads or calculates the target from values in memory or registries. These values are determined at runtime and may not be possible to obtain through static analysis. More advanced techniques have been created to counter this challenge. Shoshitaishvili et al. presents in [61] several techniques they have implemented, and describes three different types of indirect jumps.

Further difficulties arise when a jump table is used, as it is not always trivial to determine that such a structure is used. A jump table is a code structure that consists of an indirect jump with several targets. The target addresses are stored in a table, and the offset into the table is calculated at runtime every time the jump is encountered. Determining that a jump table is in used relies on analyzing the code and the structure in memory that it reads from. Additionally, even though a jump table were used in the original source code, the compiler may have altered it, through for instance optimization, such that it is no longer recognizable. Cifuentes and Emmerik [67] refers to jump tables as N-conditional branching, and presents a technique for recovering these structures from disassembled code. By slicing the code before the indirect jump is calculated and converting the instructions into a pseudo high-level language, they are able to create an abstract representation of the structure. This representation is then compared to a set of normal forms for n-conditional code, and decision is made as to if this structure is actually used and how.

To measure the quality of a control flow graph recovery method, Shoshitaishvili et al. [61] and Xu et al. [63] defines two properties: soundness and completeness. Soundness is defined as a measurement of the methods ability to recover a programs possible control flow transfers. A method is considered sound if, after the analysis has completed, all possible transfers are represented as edges in the final graph. This also includes the set of all possible targets of indirect jumps. Failing to recover every possible target of an indirect jump decreases the soundness of the method.

Completeness represents the methods ability to create a graph where every edge actually represent a control flow transfer in the original program. A method that creates a graph where every instruction has an edge to every other instruction is by definition a sound method, but is not very useful. A complete method will recover a graph where no edges represent jumps that does not exist in the original program. Therefore, a method that produces a graph which contains no edges is complete, but also mostly useless. A method must have a high degree of both soundness and completeness to be useful. This is however difficult to achieve, and a compromise is often found.

Finally, another major difficulty regarding indirect jumps, that is relevant to all static analysis of malware, is obfuscation. Indirect jumps are often used by the compiler to alter the execution flow at runtime, but may also be used by malware developers to obfuscate code and hinder automatic static analysis. One example is shown in figure 8, where the target of the indirect jump is stored



```
public start
start proc near
mov     ebx, offset loc_4073C0
jmp     ebx
start endp
```

Figure 8: Example of an unnecessary indirect jump, preventing further automatic analysis.

in register ebx. However, on the previous line, a static value is stored in ebx. In practise, this is a direct jump, but the value is moved out of the jump instruction, and into the previous instruction. A human analyst can easily determine the target of the jump, as well as the fact that there will only be one possible target in this case. Naïve automatic analysis, on the other hand, will not be able to determine the target of the jump, and will not be able to continue from this point. More advanced techniques must be employed to automatically determine the value of the registry. This example is also quite trivial, as value of the register may be calculated over the span of an arbitrary number of instructions. Both Sand [68] and Kruegel et al. [69] studied automatic analysis of obfuscated code. Sand presents several common obfuscation techniques, and how they affect the compiled code, and used a method based on dynamic analysis to detect malware. Kruegel et al. presents several techniques based on static analysis to improve disassembly of obfuscated binaries.

2.12 Graph classification

This section presents classification methods that are relevant to this thesis. The section is divided into graph matching for methods based on graph theory and matching, and graph learning for methods based on machine learning.

2.12.1 Graph matching

Methods based on graph matching are universal methods that can be used to classify or match graphs representing any object or data, not just control flow graphs. These methods directly compare nodes and edges between two graphs to measure how similar the graphs are.

Definition 5 (*Graph matching*): For graphs G and H where $|V(G)| = |V(H)|$, graph matching involves finding a bijective function $\phi : V(G) \mapsto V(H)$ that optimizes a cost function which measures the quality of the mapping. [70]

Bengoetxea divides in [5] graph matching into two subcategories: exact and inexact matching. The primary difference between these categories are that exact matching finds a perfect mapping if it exists, or no mapping at all, while inexact matching finds the best possible mapping. Exact matching tells us if one graph is exactly the same as another or not, while inexact matching tells us how similar two graphs are. These two categories can further be divided in two, by whether or not they considered the whole graph or only one or more sub-graphs when matching, as visualised in

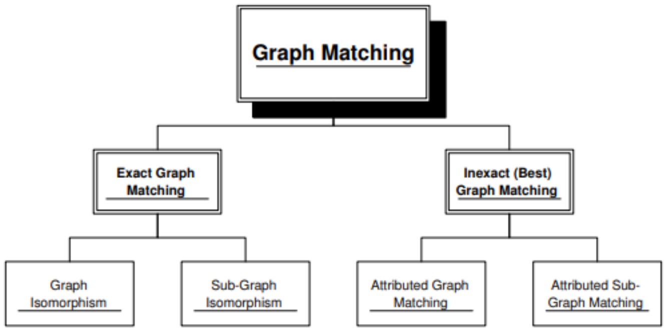

Graph Matching
Exact Graph Matching
Inexact (Best) Graph Matching
Graph Isomorphism
Sub-Graph Isomorphism
Attributed Graph Matching
Attributed Sub-Graph Matching

Figure 9: Different classes of graph matching. From [5].

figure 9.

Exact graph matching, also referred to as graph isomorphism, is of less relevance to this thesis, as it is too precise to be practical. This form of matching would be unable to detect or classify new versions of known malware, as any change would prevent a match. However, exact sub-graph matching, or sub-graph isomorphism, is still relevant. Small sub-graphs of a control flow graph recovered from malware may remain the same, even if new versions of malware are developed. A sub-graph may for instance represent a function, which may have remained unchanged when the malware was updated.

Inexact matching, on the other hand, offers the possibility to match graphs based on their similarity. As Bengoetxea states in [5], inexact matching applies when it is not possible to find an isomorphism between two graphs. This occurs for instance when there is a different number of nodes in the two matched graphs. While exact matching returns a boolean result, inexact matching usually returns a number between 0 and 1 which indicates how similar the graphs are, where 0 indicates that two graphs are identical and 1 indicates that there are no similarities [70]. Inexact matching is more relevant to this thesis than exact matching, as it can classify new malware even after it has been updated. This can be done by setting a specific threshold for similarity, and stating that any two malware must be of the same class if their control flow graphs have a similarity score greater than the threshold. As with exact matching, inexact matching may also be used with sub-graphs. Østbye used inexact sub-graph matching in [41] to classify call graphs extracted from malware.

2.12.2 Graph learning

In addition to the graph theoretic approaches presented in the previous section, it is also possible to use techniques from machine learning to classify graphs. These methods either rely on learning some structural information about the graphs, or converting the graphs into feature vectors and using traditional machine learning methods.

An example of the former case is [71]. Lee et al. presents a method that does not consider the entire graph, but automatically finds a sub-graph that contains more information, relative to other

Table 3: Instruction groups used to generate the colors [7]

Class	Description	Class	Description
Data Transfer	mov instructions	String	x86 string operations
Arithmetic	incl. shift and rotate	Flags	access of x86 flag register
Logic	incl. bit/byte operations	LEA	load effective address
Test	test and compare	Float	floating point operations
Stack	push and pop	Syscall	interrupt and system call
Branch	conditional control flow	Jump	unconditional control flow
Call	function invocation	Halt	stop instruction execution

graphs of the same class and of other classes. A training stage is used to, among other things, rank each node, and nodes of higher rank are prioritized when determining the interesting sub-graph. This is a general method, and is therefore capable of classifying any type of graphs. SUBDUE [72] is another graph learning system, which uses various techniques to learn information about graphs and find sub-structures and relational patterns. It was employed by Sand in [68] to perform malware detection using call graphs obtained from dynamic analysis.

However, many methods that attempt to learn structural information from graphs are still highly theoretical, and may not have general, functioning implementations. Some techniques counter this by converting the graphs into vectors that represent the desired features of the graphs. This vector can then be used as feature vectors with traditional machine learning methods.

One example of this, is [7], where Kruegel et al. used control flow graphs to fingerprint known worms and detect new once by matching their fingerprints. Their method relies on recovering the control flow graph from an executable binary that is transmitted over the network. Each node in the graph is then assigned a color, based on the nodes instructions. The colors are n-bit integers, where n is the number of instruction groups, and each bit corresponds to a group. The authors used in [7] 14 groups ($n = 14$), where instructions that performed similar actions or are in some way related are grouped together. The groups used in [7] are presented in table 3. The algorithm for building the color is presented in 1, and a Python implementation of the algorithm is used in appendix B. The algorithm takes as input a control flow graph node, a list of instruction groups, and the number of groups, and outputs the integer color. For every instruction in the node, if it exists in a given group, the corresponding bit in the color is set. By using colors instead of the actual instructions, the method abstracts away instructions that may have been replaced with similar instructions in new or altered versions of the worm.

The colors of each node are combined with an adjacency matrix. This procedure is visualized in figure 11, in chapter 3. This matrix is a bit matrix that models the edges of the graph, with the rows representing the originating node, and columns representing the destination node of the edge. A bit is set in the matrix if there is an edge in the graph from and to the nodes represented by the corresponding row and column, respectively. To combine the matrix and the colors, the matrix is sliced into rows, and the color of the node represented by a row is appended to the row to form

Algorithm 1 Function for building the color of a node

```
Input: $node$ is a control flow graph node
Input: $color\_groups$ is a list of instruction groups
Input: $number\_of\_groups$ is the number of instruction groups
Output: Integer $color$
function BUILD COLOR($node, color\_groups, number\_of\_groups$)
    $color \leftarrow 0$
    for all $instruction \in node$ do
        for $i \leftarrow 0$ to $number\_of\_groups$ do
            if $instruction \in color\_groups_i$ then
                $color \leftarrow color \mid (1 \ll i)$
                break
            end if
        end for
    end for
    return $color$
end function
```

a vector. The vectors are then appended to each other, to form a single vector representing the graph. By using a combination of colors and adjacency matrix, both the information stored within the nodes and structural information of the graph is included.

3 Methodology

The goal of this thesis is to develop a new method for multinomial malware classification based on static analysis that includes behavioural information, in an attempt to combine the best features of static and dynamic malware analysis. This method employs static analysis to recover the control flow graphs of malware, converts the graphs into feature vectors using various techniques, and uses the feature vectors for machine learning based classification. Additionally, graphs are pre-processed in an attempt to improve classification accuracy.

This chapter presents the theory behind the method, different ways that the method can be implemented. The actual implementations of the method chosen for the experiments in this thesis, as well as implementation details, are left to the next chapter. The proposed method is highly modular, and consists of four parts. THe four parts are control flow graph recovery, graph based pre-processing, feature vector generation and classification, as shown in figure 10.

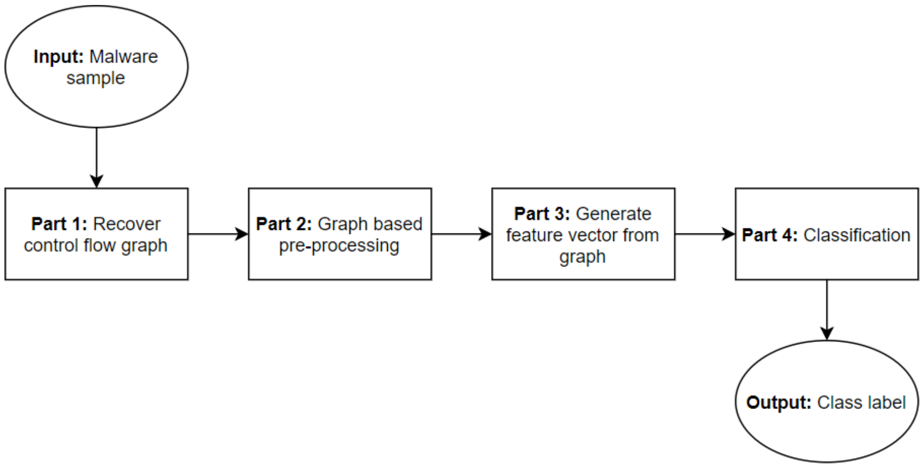

Input: Malware sample
Part 1: Recover control flow graph
Part 2: Graph based pre-processing
Part 3: Generate feature vector from graph
Part 4: Classification
Output: Class label

Figure 10: An overview of the method presented in this thesis

The input consists of malware samples, and the output is a class label. Each of the four parts can be implemented in several different ways, and an implementation of a part is referred to as a sub-method. Several sub-methods are presented for each part of the proposed method in this chapter. Information transfer between the parts is standardized, such that any sub-method of a part may be replaced with another sub-method of the same part. This simplifies development, and enables easy evaluation of performance between the different sub-methods.

This thesis is not an exhaustive test of all combinations of sub-methods. Due to time constraints, and the exponential growth of testing additional sub-methods, only some were included in testing. However, the framework developed and used in this thesis can easily be extended to include additional sub-methods. The results are therefore not exhaustive, but serve as an indication of the usefulness of the proposed method and its classification accuracy.

The chapter is divided into several sections that reflects that reflect the four parts. Each of the first four sections describes one of the parts, as well as different sub-methods suggested for the part and a comparison of them. Following this, several datasets of malware are presented and evaluated, as well as a comparison of different class-sets.

3.1 Part 1: Control flow graph recovery

The first part of the method is, as the name implies, responsible for recovering the control flow graph of a program. This part takes malware samples as input and outputs the control flow graph as a JSON object [73].

As described in section 2.11.3, control flow graphs may be represented in two different ways: one large graph representing the entire binary, or several smaller, unlinked graphs each one representing a different function or section in the binary. Therefore, for this part, two sub-methods are suggested: single and multiple graphs.

3.1.1 Single graph mode (SGM)

This sub-method aims to recover the control flow of a program into a single, large graph. Simply put, from the first node of the graph, it will be able to reach every other node in the graph. This method will use the entry point of the executable, as listed in the header of binary, and recover the graph from this point. The entry point of the executable contains the address of the first instruction, and is used by the operative system to start execution of the program after loading it into memory. By starting recovery at the entry point and including every instruction that can be reached from this point, the sub-method produces a graph that represent, from start to finish, the different control flows of the program.

To create a single, linked graph, this sub-method represent “call” and “ret” instructions as edges, the same way ordinary “jmp” instructions are represented. See figure 6 for an example of a single graph representing the dummy program in listing 2.2. Representing calling and returning from functions as edges enables the graph to show how the control flow is transferred into the function when the “call” instruction is encountered, and back out again when “ret” is reached. Functions in this sub-method are represented as sub-graphs, with a single incoming edge representing the “call” instruction, and one or many outgoing edges representing “ret” instructions. All the outgoing edges would point to the same node, as a function can under normal conditions only return to the instruction following the “call” instruction.

One major drawback of this representation is that a copy of every node in the function must be included wherever the function is called in the program. This is because the graph is not able to represent target address of the “ret” instructions based on the origin of the “call” instruction. If all instances of a function were represented by a single sub-graph, then the sub-graph would have multiple incoming edges, and multiple outgoing edges for each "ret" instruction. This is the case during normal execution of a program, where the same section of code is executed whenever the function is called. The current return address of a function is pushed onto the stack when the function is called, and popped of the stack when returning. It is not possible to represent this in the graph, and the sub-method must include a copy of the sub-graph representing the function whenever the function is called.

3.1.2 Multiple graphs mode (MGM)

This sub-method aims to recover the control flow into multiple, smaller graphs. This is done by first performing initial analysis of the program to determine function and section boundaries. Once this is done, a separate control flow graph is recovered for each function and section. That is, multiple start and end points are used when recovering the graphs, resulting in a collection of graphs that each represent a different function and section. See figure 7 for an example of a single graph representing the dummy program in listing 2.2. This collection is then used to represent the binary. As opposed to the previous method, "call" and "ret" instructions are not represented in the graphs as edges. Every function and node is therefore only included once in this representation, resulting in no redundant nodes.

Additionally, information can be assigned to functions and sections. This may be information obtained during the initial analysis of the binary or when the graph is recovered. An example of this information, is whether or not the function or section is part of the programs code, of imported from a library. This enables filtering on the function and section level, allowing us to some functions.

3.1.3 Comparison

The two sub-methods uses the same techniques for recovering the control flow graph, but represent the resulting graph in different ways. The first sub-method produces a single, large graph containing every node and edge in the executable, while the second produces multiple, smaller graphs divided on a function and section level. The single graph of the first method offers a better representation of control flow, by including how the functions interact and showing possible execution paths from start to finish. This, however, in combination with the redundant inclusion of functions, results in very large graphs that may be too big for use in classification. The second sub-method splits the graph by functions and sections, reducing redundant information and producing fewer nodes and edges, but only includes control flow information within each function and does not consider the order in which they are called . It also enables metadata-assignment on the function or section level, which enables more efficient filtering. If we were to use filtering with the first method, individual nodes would have to be filtered, which is less efficient.

3.1.4 Additional control flow graph recovery sub-methods

The sub-methods presented for this part of the method covers how the recovered graphs are structured, but leaves the details of exactly how the graphs are recovered to the implementation. In this thesis we focus on static analysis, to combine the benefits of static analysis with those of capturing behavioural information. Using dynamic analysis to recover the control flow graphs is also possible, and has the advantage of giving a more accurate representation of the actual execution. This however adds additional complexity, as described in section 2.5, and was therefore not included in this thesis. Using dynamic analysis to recover the control flow graph, and using the result with the rest of the method is left for future research.

3.2 Part 2: Graph based pre-processing

The second part of the method considers any pre-processing or adjustment that can be made to the control flow graph after recovery. While the control flow graph is a representation of the program and may directly be used for classifications, it may also be altered to improve classification accuracy. The goal of these pre-processing techniques are to remove any information that is not relevant to classification, as improving the ratio between relevant and irrelevant information will increase the differences between samples of different classes, and to structure the graphs in the best possible way for the rest of the method. To measure these improvements, the original, unprocessed graph will be included during the experiments for comparison.

Sub-methods in this part of the method implement these adjustments. They take the recovered control flow graph from part 1 of the method, and returns a new, adjusted graph. In theory, as the result of part 1 is standardized, every sub-method in part 2 could be used with every sub-method in part 1. This however, is not the case, as a given part 2 sub-method may be dependent on metadata or structural features that are only included by some part 1 sub-methods.

Three sub-methods for this part are presented in detail in the next sections, as well as a short list of other possible sub-methods that were not included in this thesis. All three of the detailed sub-methods were implemented, bot only the last two were included in the final experiments.

3.2.1 Removing library functions (RL)

A simple form of graph based pre-processing is filtering of nodes and edges. That is, to remove nodes or edges from the graphs based on some criteria. This may be attributes of the nodes or edges, or metadata covering some sub-graphs. Removing unnecessary or redundant information may improve classification accuracy. This is information that is common between most samples and does not aid in diversifying samples of different classes. Some examples of this are startup routines or imported functions from commonly used libraries. Some sub-graphs are common in all classes, while others are only observed one or a few classes.

One suggested sub-method for part 2 of the method is removing sub-graphs identified as functions from libraries. These sub-graphs can often be identified when recovering the graph from the malware, and this information can be assigned as metadata. This sub-method supports both of the two previously mentioned part 1 sub-methods, but is most efficiently used when combined with multiple graphs mode, as functions are recovered as individual graphs to which metadata can be assigned. If used with single graph mode, the metadata would have to be assigned to individual nodes.

Removing library functions reduces inter-class similarities, by removing functions or sub-graphs that are used in samples of different classes. Additionally, by removing library functions the sub-method increases focus on the code written specifically for the malware, which may further highlight the differences between the classes. One mayor drawback of this sub-method is that it in some cases can remove library functions that are only used in samples of one or a few classes, reducing inter-class similarities. How to address classification relevant information in library usage is left for future work.

3.2.2 Ordering functions (OF)

This second sub-method may be used with the multiple graphs mode (MGM) sub-method from part 1, and does not support the other proposed sub-method. This sub-method attempts to re-order the functions in the order they are called. This can be done by building a meta-graph of a sample, where the nodes represent functions and edges represent calls between them. The Dept First Search algorithm [74] can then be executed from the entry point of the sample to determine the order of the functions, by outputting every new encountered node until all nodes have been iterated over. This is done to counter one of the major drawbacks of MGM, which is that it does not consider the order of which the functions are called, only the order in which they are recovered from the sample. This sub-method makes the assumption that ordering the functions in this way gives a better representation of the behaviour of the original sample.

Ordering the functions may improve intra-class similarities, as functions may have been recovered in different orders in different samples, even though they are executed in the same order. For instance, if a sample is compile twice with ASLR [75] enabled, the two executable will have the same functions at different locations in memory. This results in different function orders if the functions are ordered by memory addresses. Therefore, by ordering the functions in the order they are called, the sub-method improves similarity between the samples of the same class.

3.2.3 Limit nodes (LN)

The third part 2 sub-method studied in this thesis is limiting nodes. This sub-method implements a naive technique for reducing the size of the control flow graphs, by limiting the number of nodes used. This technique was used by Kruegel et al. in [7], where only the first 10 nodes of each sample was used. When using this sub-method, a threshold must be set beforehand. For every sample, the nodes are counted, and any node after this threshold is reached are removed from the graphs. This sub-method may be used with both of the previously proposed part 1 sub-methods.

Determining the threshold requires additional analysis of the dataset. In this theses, the mean number of nodes in all samples of the dataset was used, as this number better represents the dataset than any arbitrarily chosen one.

This sub-method makes the assumption that the first X number of nodes contains sufficient information for classification. It does not consider whether or not these nodes are the best nodes for classification, that is, the once containing the most information. Determining the set of nodes within a sample that are best suited for classification requires additional research, and has not been studied in this thesis. Assuming that the naive approach used does not capture the nodes best suited for classification, the results obtained may be considered conservative with potential for improvements.

3.2.4 Additional pre-processing sub-methods

The three sub-methods listed above are the once studied in this thesis. Several other sub-methods may be suggested. For instance, graph theoretic approaches [76] may be applied to determine small, reoccurring sub-graphs and remove or reduce them, similarly to what is done in [72]. Techniques from information theory [77] may be applied to detect sub-graphs that offer the most information when comparing samples of different classes.

3.2.5 Multiple pre-processing sub-methods

Finally, multiple pre-processing sub-methods may be combined to form new sub-methods. The combined sub-methods are then applied sequentially to the control flow graph. This does require a degree of comparability between the sub-methods when considering which can be combined and the order they are executed in. For instance, a new sub-method can be created by combining Ordering Functions (OF) and Limiting Nodes (LN). Performing OF before LN is vital in this new sub-method, as OF would not be able to consider all functions when ordering if LN is executed first.

3.3 Part 3: Feature vector generation

The third part of the method coverts the control flow graph into a feature vector that can be used for classification. The sub-methods take the control flow graph as input, and output a vector of integers or bits that ideally contains as much classification relevant information from the graph as possible. The different sub-methods in this part present different ways a graph can be converted to a vector, but they all represent the same information. We present five different sub-methods, all of which will be compared to determine which one best captures and represents the structural information of the control flow graphs.

A graph consists of nodes and edges, both of which may be represented in the feature vector. Representing the nodes involves taking some information from each node that may be represented as a number or sequence of numbers. This information should in some way be able to diversify different programs of different classes. In this thesis, we use the assembly instructions in the nodes. Other methods may use, for instance, the number of instructions within each node or the memory address of the nodes. These attributes have not been considered in this thesis, as they may be too precise for classification. Additionally, different compilers or versions of compilers may produce executable binaries with slight differences in assembly instructs, making this information less usable for classification [68].

However, using the instructions directly suffers the same disadvantage. Common obfuscation techniques, like changing an instruction into a different one that produces the same result or reordering the instructions will greatly reduce similarities in the feature vectors. To make the method resistant to obfuscation techniques, we must abstract some details, without reducing the information to much. The method used in this thesis is colors [7], described in section 2.12.2. Each node is assigned a color consisting of 14 bits, and the bits are set based on what instructions are in the node. By grouping similar instructions, we counter obfuscation base on replacing instructions with similar once. Using 14 bits to represent nodes results in $2^{14} = 16384$ different colors. By using such a high number, we ensure that a lot of different functionality may be represented, while similar nodes are labelled with the same color. Another advantage of using colors over instructions directly is that every node is reduced to the same length in the feature vector, either as a sequence of 14 bits, or represented as a 14-bit number. In this thesis, we us the numeric representation of of colors, and a comparison of these methods is left for future work.

To represent the edges in the feature vector, we must encode either the attributes of the edges, their origin and destination, or both. Attributes of edges may be what type of jump they represent, or whether it was a far jump or not. The origin and destination may be included as pairs of nodes, or simply as the order that in which the nodes are added to the feature vector. In this part of the method, we present five different sub-methods for generating the feature vector, all of which uses colors to represent nodes, and a different way to represent edges.

3.3.1 Node-based adjacency matrix with colors (NAMc)

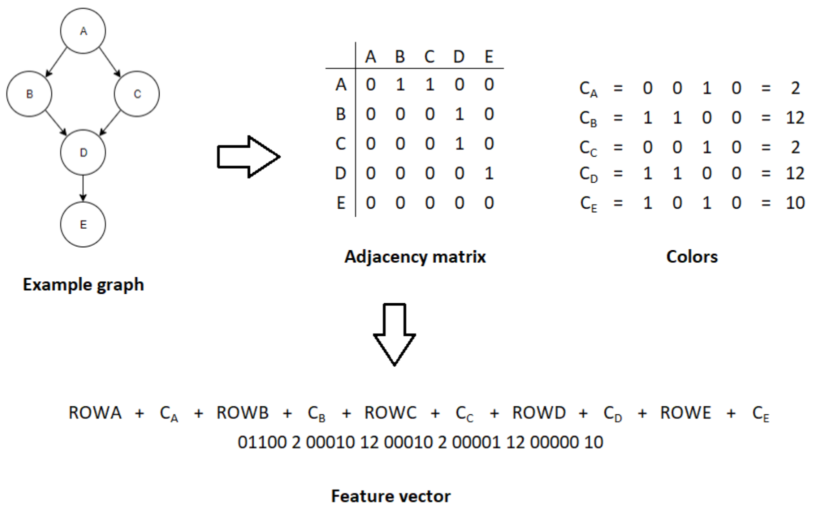


Figure 11: Visualization of Node-based adjacency matrix with color, using an example graph of nodes A-E with randomly selected, 4-bit colors. The adjacency matrix is built from the graph, and the rows are combined with the colors of each node to create the feature vector.

The first sub-method is based on the original method presented in [7], and is included for comparison. The method is described in section 2.12.2. The paper describes the adjacency matrix as a k^2 bit matrix, where k is the number of nodes recovered form the malware. As their method involves detecting malware in random streams of data, they would only perform further analysis if at least k nodes were identified. During their experiments, a value of 10 was used for k. Over the years, malware may have increased in size, and the value of k used in this thesis will be determined after an analysis of the dataset. As we are not attempting to detect malware, but classify already detected once, padding with zero nodes will be used to lengthen vectors of small malware.

Figure 11 shows a graph with five nodes, A to E. From this graph, we construct the adjacency matrix, which is a binary matrix where the rows represent source nodes and columns represent destination nodes of edges. For instance, in the top row, bit number 2 and 3 from the left are set, because there are edges from node A to both nodes B and C, but only the forth bit is set in the second row, because node B only has one outgoing edge, to node D. The color of each node is then calculated. In the example in figure 11, the colors used are randomly selected, four-bit numbers. In the final method, the colors will be 14-bit numbers, calculated from the instructions within each node. The colors are then combined with the adjacency matrix, such that the color of node A, C_A, is appended to the row representing node A, ROW_A, the color of node B, C_B, to row ROW_B, and so fourth, until all colors and rows have been combined. Finally, the feature vector is created by combining the rows into a single vector, starting with the first row.

Length of the feature vectors then becomes:

$$k^2 + k$$

where k is the number of nodes recovered from the graph. The feature vector lengths grows with the square of the number of nodes in the original graph. A result of this is that the feature vectors may become unusable for larger samples if the number of included nodes are not limited before generating the vectors.

Advantages: Previously tested, detailed inclusion of edges, vector length determined by number of nodes used.

Disadvantages: large feature vectors, sparse matrices (only two bits set per row).

3.3.2 Color-based adjacency matrix with colors (CAMc)

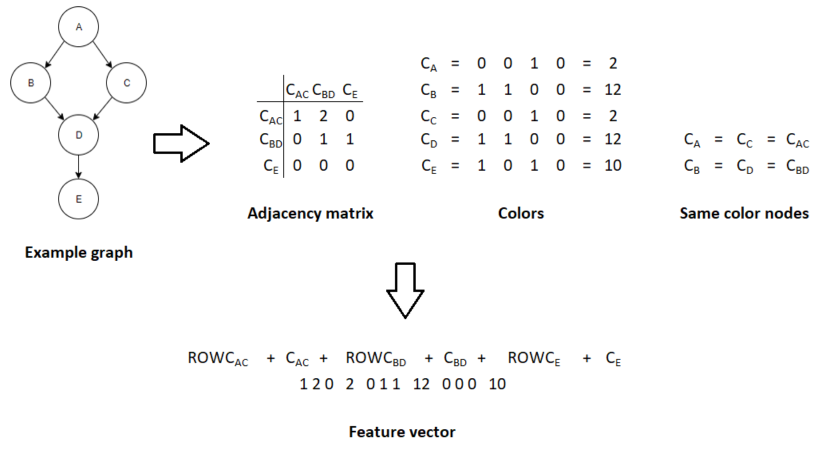


Figure 12: Visualization of Color-based adjacency matrix with colors, using an example graph of nodes A-E with randomly selected, 4-bit colors. The adjacency matrix is built by combining nodes with the same color, and the rows are combined with the colors to create the feature vector.

The second sub-method is a potential improvement on the first one, suggested by the Norwegian Defence Research Establishment. Its goal is to counter the sparseness of the previous method. Figure 12 presents a visualization of the sub-method, using an example graph of nodes A-E. Instead of building the matrix using individual nodes, the color of each node is calculated first, C_A - C_E, and the matrix is build using the colors for rows and columns. The matrix consist of integers that are equal to the number of edges from nodes with the color of the row to nodes with the color of the column. Nodes A and C have the same color, C_AC, and nodes B and D have the same color, C_BD. In the adjacency matrix, wee see that the second value on the first row is set to 2, as there are 2 edges from nodes with color C_AC to nodes with color C_BD. By building the matrix this way, it becomes less sparse, as more than two fields can be non-zero in a row. Another advantage of this sub-method compared to the previous one is shorter feature vectors. The length of the feature vectors becomes:

$$c^2 + c$$

where c is number of colors necessary to represent all nodes in the graph and $c <= k$. Usually, $c << k$, as many nodes will have the same colors, but $c = k$ is the upper limit, where every node is of a different color.

The major drawback of this sub-method is that it does not create feature vectors whose length are dependant on k. The length is dependant on the number of different colors created form the nodes of a sample, and therefore varies from sample to sample. Even if the feature vector was padded to or cut at a specific length after generating it, values of the resulting vectors would not align, as the offsets of values representing colors in the vectors differ depending on the number of different colors. The classification algorithms used in the next part compare values at the same offset in different feature vectors to each other, and may therefore end up comparing a color in one feature vector to an edge count in another. This comparison makes no sense. However, if multiple samples of the same class have the same or almost the same number of different colors, then the classifying algorithm may be able to detect that vectors of that class has colors at particular offsets, while other samples have colors at other offsets. This may be enough information for the classifier, but may not be as resistant to new samples as other methods.

Another possible drawback of this method is that information about the nodes, in the form of colors, are included twice in the feature vector. First, the color is used to build the adjacency matrix, and then, the color is appended to each row. This is further discussed in the next sub-method.

Advantages: Shorter and less sparse feature vectors ($c <= k$).

Disadvantages: Less precise representation of structural information, unaligned feature vectors.

3.3.3 Color-based adjacency matrix without colors (CAM)

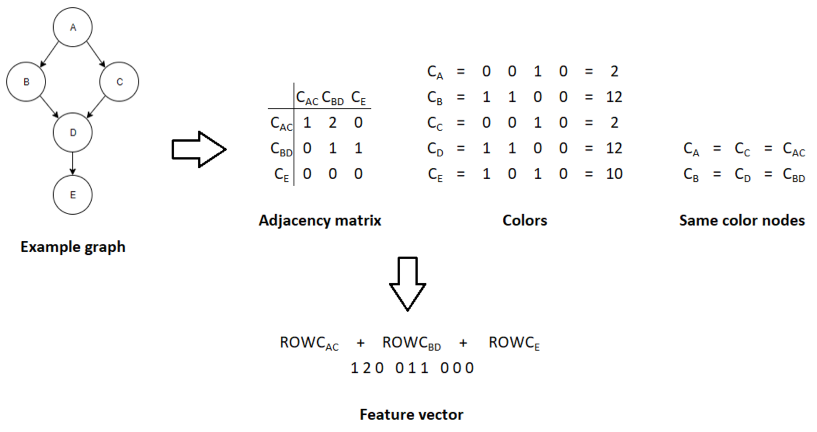


Figure 13: Visualization of Color-based adjacency matrix without colors, using an example graph of nodes A-E with randomly selected, 4-bit colors. The adjacency matrix is built by combining nodes with the same color, and the rows are combined to create the feature vector.

This sub-method is included to counter the last drawback mentioned in the previous sub-method. It is visualized in figure 13. The feature vectors are generated using the exact same algorithm as the one in the previous sub-method, but the colors are not appended to the rows when the matrix is converted into a vector. The length of the feature vector therefore becomes:

$$c^2$$

where c is number of colors necessary to represent all nodes in the graph and $c <= k$.

The goal of this sub-method is to remove the possibly redundant inclusion of colors to each row. The colors are first used to build the matrix, and then again appended to the rows of the matrix.

The primary goal of the adjacency matrix is to capture the structural information of the graph. The colors are not directly included in the matrix, and some information is therefore lost when they are not appended to the rows. We have included both this and the previous sub-method in our experiments for comparison and to determine whether the double inclusion of colors is simply redundant, or if the colors contain additional information that is not captured by the adjacency matrix.

As with the previous sub-method, this one also includes the drawback of not generating feature vectors that align between samples.

Advantages: Slightly shorter feature vectors than previous sub-method.

Disadvantages: Less precise representation of structural information, less information about colors included, unaligned feature vectors.

3.3.4 Depth first search colors (DFSc)

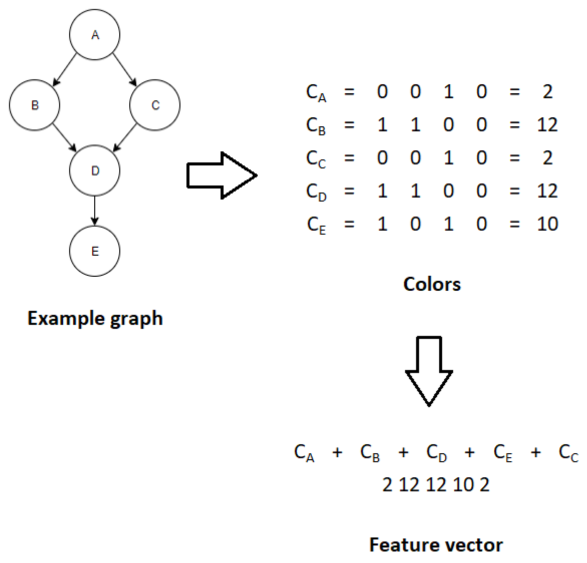


Figure 14: Visualization of Depth first search order, using an example graph of nodes A-E with randomly selected, 4-bit colors. The graph is iterated over using the depth first search algorithm, and the color of each new node encountered is appended to the feature vector.

The last two sub-methods are different from the previous three, in that they do not use an adjacency matrix to represent structural information. As with the other sub-methods, the information contained within the nodes is represented using colors. Structural information from nodes is included in the order in which the nodes are represented in the feature vectors.

This sub-method uses the Depth First Search algorithm [74] to order the nodes. This is a common algorithm for performing an exhaustive search of the nodes in a graph. It starts form a predetermined start node, and follows edges until a node with not outgoing edges is found. When a node with multiple outgoing branches is found, one of them is selected randomly and followed. If a node has no outgoing edges, the algorithm backtracks up to the last unexplored branch, and continues from there. The algorithm halts when all branches are fully explored.

The sub-method is visualized in figure 14, using an example graph of nodes A-E. The depth first search algorithm is used to iterate over all nodes in the example graph, such the order becomes A, B, D ,E and C. We then generate the color for each node encountered, and append the color to the feature vector.

The depth first search algorithm is used, as opposed to the similar breath first search algorithm [78], because the nodes are iterated over in the same order that they would be encountered during execution. Until the first node with no outgoing edges is encountered, the order produced is the same as one of the possible execution paths in the original sample. After this, when new branches are explored, the algorithm outputs smaller sequences that individually represent possible execution paths, but their combination becomes invalid. This is because, in the feature vector, there is no way to diversify between a node following the previous one, and a node being the start of a new branch.

The main advantages of this sub-method is that it produces the on average shortest feature vectors of the evaluated sub-methods, and that they does not contain any zero elements. This sub-method produces vectors of length:

$$k$$

where k is the number of nodes recovered from a sample. As previously mentioned, the feature vectors of the adjacency matrix based methods contains a lot of zero elements, because there are several node or color pairs that does not have any edges between them. This sub-method uses the color of each node, and does therefore not add any zero elements to the vector.

The two sub-methods that utilizes a color base adjacency matrix could in some cases generate shorter feature vectors, for instance if:

$$c << k$$

where c is the number of different colors and k is the number of nodes in a graph. That is, if there are several nodes in a graph but they are all labelled with a very small number of different colors. This, however, is highly unlikely, as there is such a large number of different colors that may be used to represent the nodes. Therefore, on average, the length of the feature vectors generated by this method will be shorter than in the previous.

The primary drawback of this sub-method is that it adds very little structural information to the feature vector. That is, structural information is only included in the ordering of the elements. This information may not be enough to represent the structural information, but the classifier may, in combination with the information from the nodes, still be able to differentiate different classes.

Advantages: Shortest feature vectors of the included sub-methods, easy to implement, ordering represent the execution order.

Disadvantages: Includes very little structural information.

3.3.5 Successor based (SB)

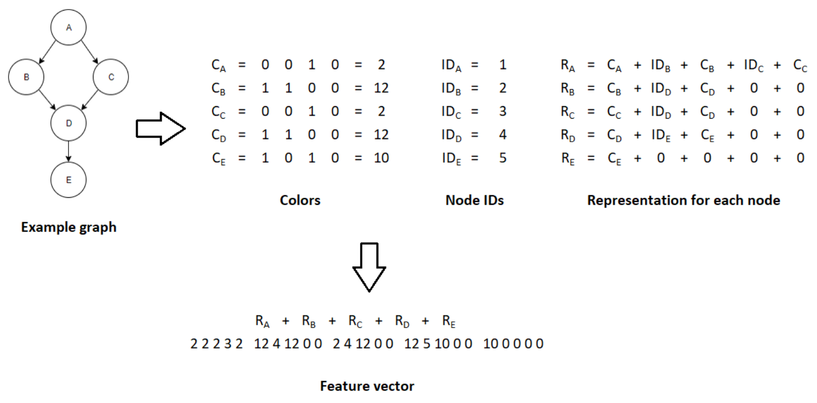


Figure 15: Visualization of Successor based, using an example graph of nodes A-E with randomly selected, 4-bit colors and sequential IDs. The graph is iterated over using the depth first search algorithm, and the color of each new node encountered, as well as the ID and color of its successors, is appended to the feature vector.

The final sub-method is based on the previous one, but attempts to improve it by adding additional structural information. It does also use the depth first search algorithm to order nodes, but includes more information with each node. Instead of only including the color for each node, it also includes information about a nodes successors. The successors of a node are the nodes targeted by outgoing edges of the original node. Each node is represented in the vector as:

$$C_{node} + S1_{id} + S1_{color} + S2_{id} + S2_{color}$$

where S1 and S2 are the successors of the node. This is visualized in figure 15. The node IDs are per sample unique numbers assigned to each node when the graph is recovered. The first node recovered is assigned the ID 0, and the number is incremented and assigned to each following node. The length of the vector then becomes:

$$k * 5$$

As described in section 2.11.1, a node in a control flow graph may have up to two successors. This makes the length proportional with the number of nodes. However, not all nodes have two successors. Therefore, when a node with less than two successors is encounter, the remaining values are set to zero. This way, the length of the feature vector remains proportional to the number of nodes recovered. Additionally, by filling in the empty values with zeroes, we keep alignment between different feature vectors.

This sub-method represents structural information in two ways. Firstly, some information about the structure of the control flow graph is included in the order that the vector is built. The second way structural information is included is in the inclusion of successors following each node. Adding information about a nodes successors to each nodes representation in the feature vector may help diversify similar nodes, and adds information about the relationships between the nodes in the graph. Additionally, the inclusion also implicitly states how many successors a node has, as no valid node will have a color corresponding to the value zero, as a color value of zero indicates that the node would contain no instructions. Finally, by including the node id of the successors, we strengthen the similarities in the feature vector where a node have the exact same successors in different graph. This also helps diversify relationships where the colors are the same, but the but the original nodes may have been completely different.

This method inherits some of the advantages of the previous one. It generates shorter vectors than the adjacency based sub-methods, and less sparse once. As opposed to the previous sub-method, this one can contain zero elements, for nodes that only have zero or one successor.

The main drawback of this method is that it may contain some redundant information with regards to the inclusion of both successor IDs and colors. The first drawback comes from the fact that successor IDs and colors both represent the same information and may therefore be considered redundant. They do however represent the information in different ways, with IDs giving a more specific representation, and colors a more general. The IDs may therefore help diversify similar samples, while the colors may improve generalization with regards to classifying new versions of a sample. Another drawback of this method is that it may be too dependent on the order of nodes. If a function or sub-structure is moved to a different location in in a new version of a malware, then it will have a completely different location in in the feature vector, making finding the similarities between the vectors more difficult.

Advantages: Shorter, less sparse feature vectors than AM-based sub-methods, structural information is included both small scale (successors) and large scale (order), ordering represent the execution order.

Disadvantages: Possibly redundant information.

3.3.6 Summary

The first method is included as it was presented in [7], for comparison. The second and third sub-methods are suggested improvements of the first method, aiming to achieve a similar accuracy using shorter, less sparse feature vectors. The final two methods are fundamentally different, and are included to determine how well adjacency matrix based methods compare to simpler more linear methods, that generates even shorter feature vectors. The fifth sub-method is a more advanced version of the forth, included to determine if additional, small scale information can improve classification accuracy.

3.3.7 Additional sub-methods

The sub-methods described in this section are the once that were evaluated in this theses, and are by no means exhaustive, but used to illustrate a potential. There exists several other sub-methods that was not included in this thesis. For instance, the last two methods used depth first search to order the nodes. These can be implemented using other graph search algorithms, such as breath first search [78] or best first search [79]. The former will order nodes at the start of the graph to the start of the vector, but looses the order of execution. The latter uses some heuristic evaluation function to determine which successor is the most interesting, and follow that branch. This function can for instance depend on the color of the next node or the number of nodes after it. Additional sub-method may use a completely different algorithm to include structural information. An in depth analysis of feature vector generation methods is left for future work.

3.4 Part 4: Classification

While graph based classification methods exist [72][71], that are able to classify graphs without converting them to feature vectors, these were not included due to limited support for multinomial classification, or lack of working reference implementations. In this thesis, we therefore focus on classification methods that accepts feature vectors.

Four different classifiers were evaluated in this thesis. The classifiers were implemented as sub-methods of part 4 of the proposed method. Each classifier is therefore interchangeable with the others, and can be used with any part 3 sub-method. The classifiers are described below, after a description of how classification accuracy is calculated.

3.4.1 Calculating classification accuracy

Classification accuracy is used as a metric to determine the performance of a trained classifier and to compare the performance of different classifiers. It is calculated as:

$$Accuracy = \frac{\text{Number of correctly classified samples}}{\text{Total number of samples}}$$

3.4.2 Random Forest (RF)

The random forest method is designed to improve the classification accuracy of tree-based learning algorithms [80]. Instead of only generating a single tree, multiple trees are generated. Each tree is generated using a different, small set of features from the feature vector. The number of features used for each tree varies, but is often set to $\lfloor \log a \rfloor + 1$ where a is the number of features in the feature vector. Every sample in the training set is used to train every tree in the forest. When classifying new feature vectors, each tree process its set of features and outputs a class. The output forms a class probability distribution, and the majority vote is selected as the most probable class for the sample.

A random forest models generalization error decreases as the number of trees increases [81]. Individual trees may be prone to overfitting. This is countered by correlating the results of multiple trees.

Random forest is a well researched method, and is considered to be on par with state-of-the-art algorithms with regards to classification accuracy and generalization [80], as well as efficiency on large datasets [82]. Its main drawback comes from its complexity and the incomprehensibility of its decisions. The model may consist of hundreds of trees, and interpreting how a decision was formed between them is a difficult task.

3.4.3 Naive Bayes (NB)

The Naive Bayes classifier is a probabilistic classifier that assumes the conditional independence of features with regards to class [80]. It can be derived from Bayes rule [83], and the following equation is taken from [80]:

$$P(C_k|V) = P(C_k)\prod_{i=1}^{a}\frac{P(C_k|v_i)}{P(C_k)}$$

Where $V = v_0, .., v_a$ are assumed to be conditionally independent features, C_k is a given class. The equation states that the probability of a class given a set of features is equal to the probability of that class multiplied by the product of the probabilities of the class given the individual features divided by the probability of the class. Training a Naive Bayes classifier involves calculating the probabilities of the individual features, or the right-hand side of the equation. This knowledge is then stored in a tabular form, where the values approximate the prior class probabilities C_k and the conditional class probabilities $P(C_k|v_i)$. These values are then used to classify new samples, by calculating $P(C_k|V_n)$, where V_n represents the new sample, using above equation. The class with the highest probability given the features is output as the class of the sample.

The assumption that features are conditionally independent does not always hold true in the real world. H. Zhang argues in [84] that conditional independent features are not required when using the Naive Bayes classifier, and that it is the distribution of dependencies between all features over classes that affects the classifier. He goes on to prove that this is the case, and that this is the reason why the classifier works well in real life scenarios, where the features are not completely independent.

3.4.4 Support Vector Machine (SVM)

Support Vector Machines (SVMs) are several supervised learning methods used for both classification and regression. This thesis uses classification, and have therefore focused on these methods. SVMs use all features of samples when performing classification.

A SVM model is trained by determining the optimal hyperplane for separating samples of two different classes. If the classes are linearly separable, then there may exists several hyperplanes that separates the classes. The optimal hyperplane is defined by the samples that are nearest to it, referred to as support vectors, and is defined as the separating hyperplane that is the furthest away from all the support vectors. If more than two classes are included, then the problem is devided into several sub-problems, each one defining a optimal hyperplane for one of the classes against the combination of the others. New samples classified as the class which maximizes the decision function from equation 10.52 in [80].

A transformation function can be used to handle cases where two classes cannot be linearly separated by a hyperplane [85]. The transformation function is a non-linear function that maps the samples into a high dimensional feature space. The optimal hyperplane is then constructed in this space, using the same technique as above.

3.4.5 Multi-layered perceptron (MLP)

A Multi-layered Perceptron (MLP) is a type of Artificial Neural Networks where hidden layers are added between the input and output layer to model non-linear relationships [80]. Models are built using neurons, separated into layers, which interacts with each other using weighed connections [86]. Each neuron in one layer is connected to every neuron in the previous and the next layer. The first layer is the input layer, the final layer is the output layer, and the one-to-many middle layers are the hidden layers.

Information is encoded into a trained MLP model as the weights of the connections. Models are trained using a method called backpropagation of errors. Initially, the model is created with random weights. Labelled training samples are then passed through the model, and the difference between the resulting class and the target class is calculated. The weights are then adjusted to reduce this difference. When performing classification, new samples are passed through the model in the same way to obtain the predicted class.

As with Random Forest, a major drawback of MLP is the complexity of the models [87]. A trained model may have several hundred neurons and connections, and determining how each one of these contributed to the resulting decision is quite difficult. Another drawback of MLP is the number of meta-parameters that must be set before creating a model. These include, among others, designing the topology of the network: determining the number of hidden layers to use, and how many neurons to use in each one.

3.4.6 Additional classification sub-methods

As with the other parts, additional sub-methods can be studied in this part of the method. These may implement other classification methods, or use different meta-parameters for the once used in this thesis.

3.5 Note on combining sub-methods

The proposed method is designed to be modular, using standardized interfaces between each part. This was done to so that different sub-methods can be developed and combined with other sub-methods, without changing the other sub-methods. This enables easy development and comparison of different sub-methods. However, this is not possible in all cases. For instance the part 2 sub-method Ordering Functions requires that Multiple Graphs Mode is used in part 1, as the functions cannot be ordered without the graph being separated into functions. Another example in part 2 of this is Removing library functions, witch requires that either nodes (when combined with Single Graph Mode) or functions (when combined with Multiple Graphs Mode) are labeled as to whether or not they are from a library.

This is a slight limitation of the proposed method, but can easily countered during the implementation of the sub-methods. For instance, if we wish to use Removing library functions in part 2, we can easily modify the sub-method used in part 1 so that it includes the required information as labels. These labels can then be ignored when combined with part 2 sub-methods that does not require the information.

3.6 Datasets

As with most methods relying on machine learning, a dataset is required to evaluate the performance of one proposed in this thesis. Features are extracted from samples in the dataset, and used to train a model such that it represents the relationship between the features and the class of each sample. This dataset must consist of malware samples that gives a representation of malware found on the internet.

There currently exists several datasets of malware on the internet, many of which have previously been used in research. In the next sections, we present some of the often-used datasets and offers a short evaluation in the context of this thesis.

3.6.1 Evaluation of existing datasets

Before evaluating the different datasets, some criteria must be set that the sets must fulfill. For a dataset to be used in this thesis, it must be large enough, such that it can be used in experiments, must be representative of the real world, it must be up to date, it must be dividable into several classes, and each sampled must be pre-assigned labels corresponding to their class.

VirusShare.com

This is a personal project, developed for use in research. Malware is collected and uploaded to the site by the author, and samples are stored with a report from VirusTotal [88]. As of March 2019, the site contains over 33 million samples of various file types.

Advantages: Large, several different classes and filetypes.

Disadvantages: Requires filtering, less professional source, unlabelled.

theZoo

theZoo is an open repository of malware samples and source code were everyone can submit additional samples.

Advantages: Easily available, many different classes and filetypes, gathered from many different sources.

Disadvantages: Requires filtering, less professional source, small, unlabelled.

CCC DATAset 2009

The CCC DATAset 2009 was created for use in the anit-Malware engineering WorkShop (MWS) [89]. The malware was collected using 94 honeypots over the period of one year.

Advantages: Highly studied dataset, professional source.

Disadvantages: Small, limited to malware types that attacks the honeypots used.

Microsoft Malware Classification Challenge (BIG 2015)

To encounter the increasing number of existing malware, Microsoft launched in 2015 a competition to crowd-source development of malware classification methods. For this competition, they created a dataset of malware consisting of samples from nine different, known families. The dataset is large, almost 0.5TB when uncompressed, and have been used a lot in research.

Advantages: Large, labelled, several different classes, professional source, used in other research.

Disadvantages: Samples altered to not be executable and is therefore dependant on proprietary pre-processing.

Testimon Research Group malware dataset

This dataset was created by the Testimon Research Group at NTNU during April and June 2015. The samples were gathered from other datasets, including VirusShare.com. The dataset consists of over 400000 malware samples. Microsoft were used label each sample with a type and a family.

Advantages: Large, labelled, several different classes, professional source, used in other research, unmodified samples.

Disadvantages: Limited to one filetype.

3.6.2 Dataset used in this thesis

For experiments conducted in this thesis, we selected the dataset created by the Testimon Research Group. The main advantage of the this set over the dataset created by Microsoft is that the samples are unmodified, giving a more accurate representation of the real world. The other sets are either too small for use in experimentation, or requires quality control and filtering to be used. By using the selected dataset, we rely on the filtering conducted by the Testimon Research Group.

More specifically, we will be using a sub-set of the dataset, using 9823 samples, divided into ten classes. This set contains an almost even distribution of samples between the classes, with nine classes containing 1000 samples each, and the final one containing 823 samples. Further statistics about the dataset used for experiments are presented in sections 4.1.1 and 5.2.

Banin and Dyrkolbotn used in [43] a similar sub-set, using the same ten classes, but about 100 samples of each. They constructed n-grams of memory access operations (reads and writes), recorded during dynamic analysis, and used these n-grams as feature vectors for classification. Using this method, they achieved a classification accuracy of 84.5%.

3.7 Class-sets

When creating the dataset used in this thesis, the Testimon Research Group used Microsoft [1] to assign labels to each samples. The primary reason for this is that the naming system used by Microsoft contains both the type and the family of a sample. Each sample were therefore assigned both a type and a family. Both [1] and [43] defines malware types as groupings based on what the malware does, while malware families is based on how the malware does it. Banin and Dyrkolbotn performed in [43] a comparison using the different labels. They found that they achieved better classification accuracy when using families as opposed to types, and argued that this is because that characteristics covered by families are more present in the samples. That is, it is easier to classify samples that does the same thing in the same way (families) than it is to classify samples that does the same thing, but in different ways (types).

Determining which class-set to use depends on what information we wish to obtain from the method. What information is considered interesting or relevant depends on context. Malware types may be more interesting when handling incidents, as the capabilities of the malware will help in decision making. Malware families may be more interesting when determining the origin of an attack or who developed the malware. In general, this is a difficult decision, and is not the focus of this thesis.

In this thesis, families were used when performing classification. This is primarily because our method relies on recovering structural information using static analysis, and samples within a malware family have a more similar structure than those of the same type. This is in turn because samples in a malware family performs the same action in the same way, and therefore has the same or similar structure. Samples of the same type may perform the same actions, but uses different methods, resulting in different structures. This makes our method more capable of classifying families, which is why this class-set was used in this thesis. This is also supported by [43], where the authors achieved a higher classification accuracy with families than with types. Further research may be conducted to evaluate classification accuracy when using the proposed method to classify malware types. It is also worth nothing that the proposed method is class-set agnostic, and will use whatever labels are assigned to the control flow graphs when they are recovered in part 1.

4 Experiment

This chapter presents the practical component of the thesis and experiments conducted. These experiments were designed to evaluate the proposed method, using a realistic dataset. However, due to the modularity of the method and problems encountered during development, some of the proposed sub-methods was not included in the experiments. The experiments focus on evaluating the original method by Kruegel et al. [7] and comparing it to our suggested improved versions, as well as some fundamentally different versions, based on the depth first search algorithm. Each implementation of the proposed method combines one sub-method from each part, and a total of 72 different implementations were evaluated.

This chapter starts by covering how colors were implemented and used, before describing some problems that were encountered when designing and implementing the method. A preliminary experiment was conducted during the thesis to evaluate how our implementation of the original method performed, and how well the original method performed with a modern dataset. In this preliminary experiment, we performed binomial classification on a dataset of benign software and malware, and the experiment setup and results are represented, as well as a comparison to the original results. Finally, the experiment setup is presented, including a description of the different experiments that were conducted, and the parameters for these.

4.1 Colors

A key part of the method presented in this thesis is the use of colors to represent the information stored in the nodes of control flow graphs. In this thesis, we use colors in the same way as Kruegel et al. in [7]. Assembly instructions are divided into 14 groups, and each group represents a bit in a 14-bit color. Each node is assigned color based on its instructions. Colors are further described in detail in section 2.12.2.

Additionally, we use the same instruction groups as Kruegel et al., listed in table 3. The groups was created such that instructions in a group may be replaced by other instructions in the same group. However, the authors did not include which instructions were assigned to which group. Therefore, the groups used in this thesis may not contain the exact same instructions as the once used in the original paper, but are based on an educated guess from the descriptions in [7]. The groups created and used in this thesis are presented in appendix C. This difference may slightly affect comparison between our methods and the one used by Kruegel et al.

Due to the continual development of processors, there exists several thousand assembly instructions, and the number increases with every new processor [54]. This makes including every existing instruction in the groups practically impossible. The groups were therefore created by listing every unique instruction in the recovered control flow graphs for the malware in our dataset. Were new

Table 4: Some instructions that fit into more than one color group. Descriptions from [8]

Instruction	Description	Possible groups
ADDPD	Add Packed Double-Precision Floating-Point Values	Arithmetic, floating point operations
CMPPD	Compare Packed Double-Precision Floating-Point Values	Test, floating point operations
SETcc	Set Byte on Condition	Data Transfer, Test
XADD	Exchange and Add	Data Transfer, Arithmetic

malware to be analyzed, we would have to place every not previously observed instruction in one of the 14 groups before analysis. This is a small limitation of the method when used in real life scenarios, and can be countered by adding additional instructions before deploying the method or developing a method for automatic parsing of developer manuals, e.g. [54], but this has been left for future work. These groups are, however, sufficient for the experiments performed in this thesis.

One major difficulty was encountered when creating the groups: several instructions fit into more than one group. Some examples of this, with a short description and the possible groups they fit into, are listed in table 4. These are complex instructions, that usually perform more than one action when executed, or perform their action on specific datatypes. These instructions were put into the most common of the possible groups that they could fit into. That is, the groups that most often appear in the dataset. This way, they have the least impact on the color as possible, reducing bias from incorrectly placed instructions. Additionally, we increase the likelihood of other instructions in a given, complex instructions group existing in a given node, reducing its impact on the resulting color. Two other ways to handle complex instructions are placing them in multiple groups, or ignoring them. In the first way, a complex instruction is placed into all the groups that it fit into, and when the instruction is encountered, the corresponding bit to all those groups are set. In the second, the complex instructions are ignored, and no bit of the color is set when they are encountered. Evaluating the effect of these methods on the resulting feature vectors is left for future work.

In this thesis, the colors are interpreted as 14-bit numbers, and not simple bit-sequences. This may cause the order of the groups to impact the result, as the presence of some groups of instructions have a higher impact on the resulting color than others. We used the same groups as [7], and therefore also used the same order. Another possible way would be to order them from least likely to most likely. This way, instruction groups that occur more commonly have less impact on the resulting color, while less common instructions have a large impact. According to information theory [77], rarer occurrences offer more information that common once, and this ordering enables the color to highlight these rare occurrences.

Only one set of groups were used in this thesis, and no other ways were tested for comparison, due to time restrictions, and to enable comparison to the original method in [7]. Further studies will be required to determine whether or not there exists better ways to group the instructions. These studies should consider: the number of groups, what set of instructions each group represents, which instructions are placed into which group, and the order of the groups.

4.1.1 Color frequencies in the dataset

This section presents an analysis of color frequency distributions of the different classes in the dataset. The distributions are shown in figure 16. Each chart represents one of the then classes in the dataset. The horizontal axis represent the numeric values of colors in increasing numeric value, from 0, which is the color of a empty node with no instructions, to 2^{14}, which is the color of a node with one or more instructions from each node. The vertical axis represent the percentage of nodes with a given color in all samples within the class. By using percentages, as opposed to total number of nodes with the different colors, we can more easily compare different classes. The vertical axis of each chart stretches from 0% to 10%, as no single color represented more than 10% of the nodes in any of the classes.

The first thing that becomes apparent when analyzing the charts in figure 16 is that most nodes in the dataset have the same or similar colors, and that most classes have the same clusters of colors within them. These clusters form because instructions in some of the groups are more common than others, and are often present in nodes where the less common instructions also appear. While these clusters appear in all classes, their magnitudes and local distributions vary. Some, like family 5 (Renos) and family 10 (Zlob), have a few narrow spikes, while others, like family 2 (Hupigon) and family 7 (Vb) are more evenly distributed into several spikes. These differences show that the colors are able to capture some information about the different classes that can be used in classification.

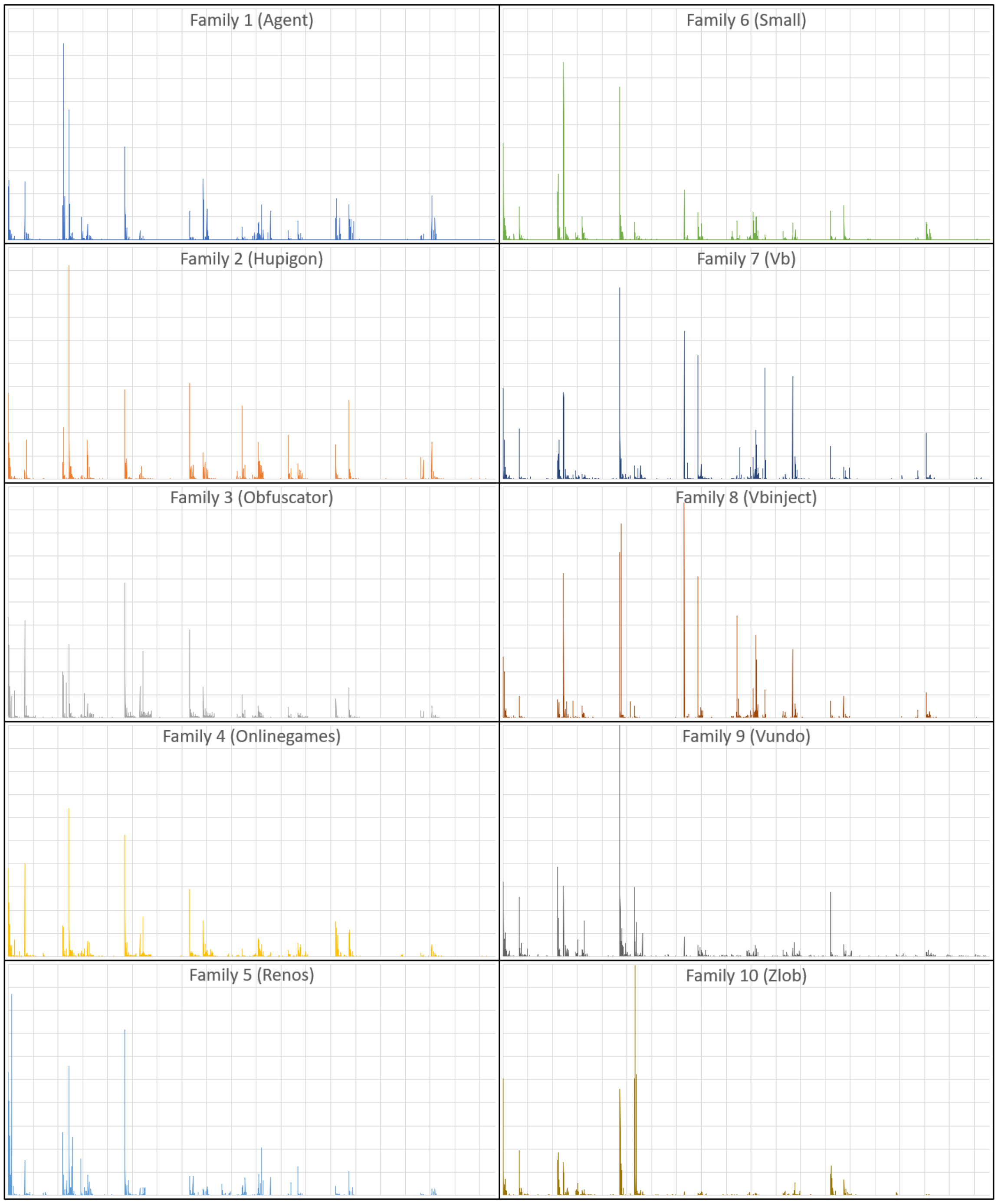


Figure 16: Color distributions of the different classes in the dataset. The horizontal axes represent numerical colors, from 0 to 2^{14}. The vertical axes represent the percentage of nodes with a given color, from 0% to 10%.

4.2 Problems encountered

This section highlights some of the problems that were encountered during development and implementation of the proposed method. It also describes why some sub-methods were excluded from the experiments.

4.2.1 Problems with single graph mode

Section 3.1 describes how control flow graphs can be recovered from malware in two different ways: single graph and multiple graphs. Initially, during experimentation, both methods were tested. To recover control flow graphs in single graph mode, the malware analysis framework angr was used. angr is a Python framework developed by Shoshitaishvili et al. [61], originally for use in the DARPA Cyber Grand Challenge [90]. It employs several binary analysis techniques, and is able to recover the control flow graph from a program using static analysis. As a python framework, it also support automation, making it a great fit for these experiments.

However, initial experimentation with control flow graph recovery showed problems with angr. To save time, the initial experiments were conducted using a sub-set of the dataset, using only 1000 samples, 100 from each class. The experiment setup, and results are presented in table 4.2.1. For comparison, the setup for and results of multiple graphs mode, using IDA Pro [62], is also included.

We encountered two major problems when using angr to recover the control flow graph. The first problem was the amount of time required to process the dataset. The virtual machine used to conduct the experiment was assigned eight virtual CPU cores and eight GB of RAM. As Python, and therefore angr, is single threaded [91] and can therefore only use a singe CPU core at a time, we attempted to use multiprocessing to process multiple samples in parallel to reduce time. However, during the experiments, we noticed several major memory usage spikes. As single memory spike was not problematic, but when more than one happened at the same time, the operative system would kill some of the processes performing the analysis. This unpredictable use of memory made parallel execution impractical. Processing the 1000 sample dataset without parallelization took over 20 hours, rendering angr unusable with the larger dataset. Recovering the same dataset with IDA Pro took about 5 minutes, although this was done using 8 IDA Pro worker-processes in parallel, as IDA Pro do not have the same memory problems as angr. The script used to execute IDA Pro is presented in appendix A.

The other major problem with angr is that it is unable to analyze several samples, simply outputting an error message and exiting. Several of these errors have been reported to the developers, but has not been fixed at the time of this project. These errors are referred to as explicit fails in table 4.2.1. In addition to these, two implicit fails were noted where analysis of the samples terminated without output. These fails were most likely caused by memory usage, where the OS terminated the processes to prevent running out of memory.

Due to these problems with angr, we did not include single graph mode in the experiments. This mode could have been implemented in other ways, for instance using Ida Pro and its Xref feature [92] to couple together the functions. However, due to the complexity of this task and the time constraints of this project, this solution was not attempted, and was moved to future work.

Table 5: Setup and results for initial control flow graph recovery experiments.

	Single graph mode	**Multiple graph mode**
Setup		
Operative system (VM)	Kali Linux 2019.1 (64-bit)	Windows 7 (64-bit)
Scripting language	Python 3.6.8	Python 2.7
Software used	angr (version: 8.18.10.25)	Ida Pro 7.2
Number of CPU cores	8	8
RAM	8 GB	8 GB
Results		
Dataset size	1000	1000
Time	~ 20 hours	~ 5 minutes
Successfully analysed	930	1000
Explicitly failed	68	0
Implicitly failed	2	0

4.2.2 Minimum analysis requirements

During experimentation, we noticed several samples whose control flow graph contained very few nodes. By analyzing the dataset, we assumed that these samples fell into one of two categories: obfuscated and packed. Obfuscated samples contain some obfuscation technique that prevents or hinders analysis, resulting in an incomplete or incorrect control flow graph. A common example of this was presented in figure 8, where the program makes an indirect jump by jumping to an address stored in a registry. The other category is packed samples, where the original malware has been compressed or encrypted, and only the decompression/decryption is visible in the sample. It is impossible to analyze the original malware without either using dynamic analysis, or unpacking the malware using some other method, and performing static analysis on the result.

In both cases, control flow graph recovery fails to produce a complete and accurate graph, and the resulting graph does not represent the original sample. Therefore, they can not be used with classification. Features that does not represent their source cannot be used to train a model that represents the real world.

What is common between these two categories is that the recovered graph often contains very few nodes, as analysis fails to find the next node, or the rest off the nodes are compressed or encoded. Therefore, we can exclude samples were the control flow graph contains few nodes. C. Krugel et al. used the same technique in [7], where they excluded any graph where they were unable to recover at least 20 nodes. They did however include other factors in their experiments, to discover malware in raw network traffic, which probably increased their required threshold. Determining the exact threshold for number of nodes required for a sample to be included requires additional analysis. Figure 17 shows the percentage of samples that will be included as the threshold is increased. If the threshold is set to zero nodes, then 100% of the sample will be included. A threshold of 6 nodes includes about 70% of the samples in the original dataset. If the threshold is

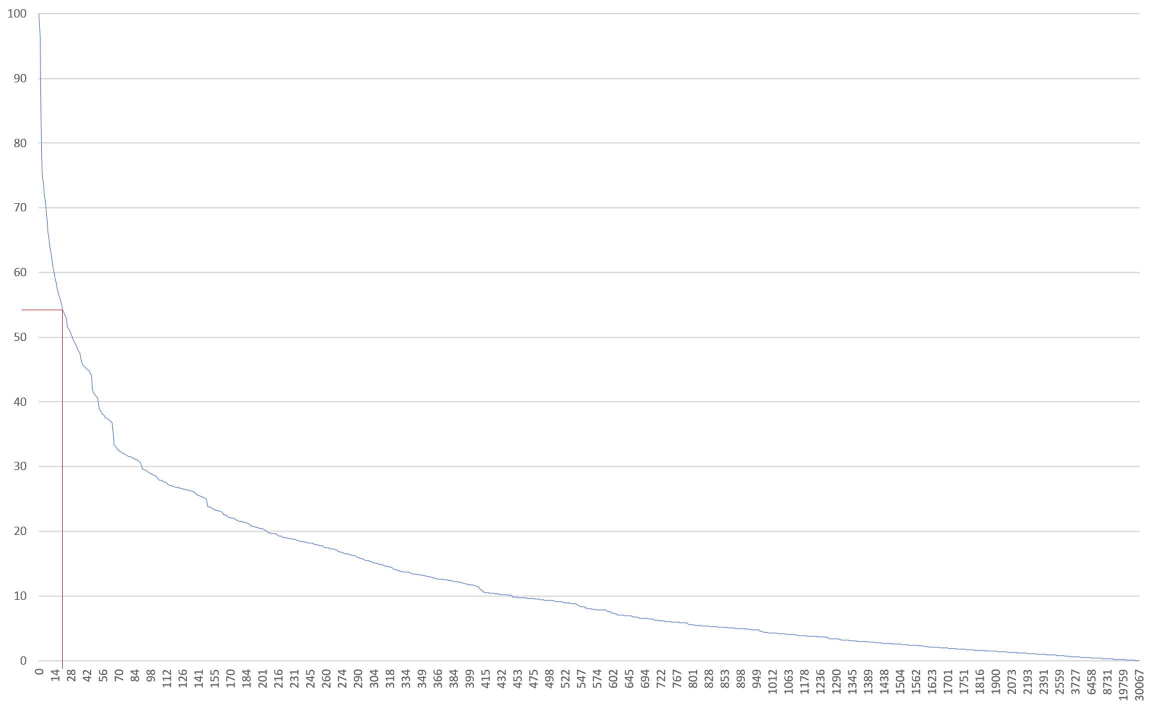


Figure 17: Percentage of samples (vertical axis) included as the node requirement (horizontal axis) increases. The red lines represent the threshold of 20 nodes used in this thesis, resulting in about 55% of the samples being included.

increased to 20 nodes, then only about 55% of the samples will be included. In the experiments conducted in this thesis, a threshold of 20 nodes were set. This removes the majority of samples where the control flow graph recovery failed or produced a low number of nodes, and ensures that every sample is represented by a not insignificant graph. By using this threshold, we remove about 45% of the samples in the dataset, which is a significant number of samples, but we still include over 5000 samples for the experiments. The exact number of samples included, and the distribution between the classes, is presented in section 5.2.

Another method for removing packed samples is checking the names of the sections in a sample. The popular packer UPX [93] sets then names of the sections to some form of “upx”. We can therefore remove all samples packed using this packer by checking section names and remove samples with section names that contains some form of “upx”.

4.2.3 Problems identifying imported functions

One of the sub-methods suggested in section 3.2.1 for part 2 of the method is removing library or imported functions and sections. Early experiments showed that this had very little effect on the resulting graphs. Table 6 shows a comparison of control flow graph recovery statistics, with and without functions identified as from libraries. We see that only 40411 nodes, or about 11%, were removed from the complete dataset.

Table 6: Number of nodes and edges before and after removing library functions and sections.

	Full samples		Removed libraries	
	Nodes	Edges	Nodes	Edges
Minimum	1	0	1	0
Maximum	30067	39661	29813	39633
Mean	243.39	276.45	239.51	276.42
Total included nodes	358065		317654	

Fewer nodes than expected were removed from the resulting control flow graphs. Two possible reasons were identified for this. Firstly, IDA Pro [62] was used to determine whether or not a given function was from a library, and included this information as metadata of the functions control flow graph. IDA Pro uses FLIRT [94] to identify library functions , which performs signature matching of the code within each function against a database of known signatures of library functions. If a match is found, then the function is determined to be from a library. This may be a source of error, if functions are incorrectly identified.

Secondly, a common obfuscation technique is dynamic imports, where the libraries imported by the malware only becomes apparent during execution. Dynamically imported functions are not discovered by IDA Pro, and can therefore not be removed.

Removing library functions were therefore not included in the experiments in this thesis, as it did not result in a significant difference. Further analysis of library functions compared to normal functions is required to create a better method for identifying these functions.

4.2.4 Recovering program sections

During initial testing, several samples were discovered that did not contain any functions. This caused control flow graph recovery using multiple graph mode to fail, resulting in no nodes or edges. Functions are a high-level programming concept created to simplify software development, and are not required for code to be executed [95]. As described in section 2.10, a common obfuscation technique is innlining, were function code replaces calls to the function. If innlining is applied to the whole program, the resulting obfuscated program contains no functions.

Analysis of program sections were included to counter malware with no functions. If no functions are found in a sample, we attempt to recover control flow graphs from the sections. Sections containing executable code are identified using the flags in the section headers. A control flow graph is then recovered from each section, using the depth-first search algorithm to iterate over every node and edge starting at the start of the section. Additionally, the entrypoint of the program is used to recover a control flow graph, if the code it points to is not included in one of the previous graphs.

The inclusion of section in the analysis counters some obfuscation techniques, and enables us to include samples in our experiment that contains no functions. We therefore included this technique is our experiments as an alternative way to recover the control flow graphs from malware samples and to counter some of obfuscation techniques.

4.2.5 Problems with feature vector lengths

When considering the lengths of the feature vectors generated by part 3 of the proposed method, two problems became apparent. Firstly, the length of a feature vector is dependent on the sample it was generated from, and the method therefore produced vectors of different lengths. This is a problem, because most classification methods require that all feature vectors are of the same length. Secondly, several samples resulted in feature vectors that were too long to be used in classification.

To solve the first problem, we can either truncate all vectors to a specified length and pad with zeros all shorter vectors, or pad all vectors with zeroes such that they become the length of the longest feature vector. Both methods have drawbacks, the first results in a loss of information, while the second results in very long feature vectors. In the experiments, truncation was used, as, as highlighted in the second problem, the longest feature vectors were too long to be used.

There are two ways to truncate the feature vectors. Either, the generated feature vector is truncated after it has been generated, or the number of nodes recovered from the samples is limited to a fixed value. For some sub-methods, specifically the once based on depth first search, these should result in minimal differences, as the nodes are represented in the same order in the feature vector. For the adjacency-matrix based sub-methods, this does however result in a difference. These include information about later nodes early in the feature vector, and excluding these later nodes during feature vector generation will produce a significantly different vector. Additionally, for the color-based adjacency matrix methods, the vector length is dependent on the number of colors within each graph, rather than the number of nodes. This results in a necessity to pad the generated feature vectors with zeroes after generating them, such that they all become the length of the longest vector.

For comparison, both truncation before and after generating the feature vectors were included in the experiments. When truncating after generating the feature vectors, shorter vectors are padded with zeroes. When truncating nodes, nodes with no instructions and no edges were inserted to pad the smaller graphs to the specified number of nodes.

4.3 Preliminary experiment

To get an initial idea of the expected performance of the method, a preliminary experiment was conducted. The method presented in this thesis builds heavily on the method presented by C. Kruegel et al. in [7]. However, their experiments included detection of polymorphic worms among normal web traffic, adding additional complexity to the binomial classification problem that is malware detection. Additionally, the malware testes was limited to polymorphic worms, and not general malware.

The preliminary experiment aims to reproduce the method used in [7] for worm detection, and evaluate it with a simple binominal classification problem. This experiment evaluates the methods ability to detect malware from benign software. By reducing the method to this classification problem, and removing factors that are not relevant to our experiment, we are able to obtain an initial assessment of how well the method will performed.

4.3.1 Dataset

The dataset used in this experiment is a sub-set of the same dataset from which the dataset described in section 3.6.2 is a sub-set of, the Testimon Research Group malware dataset. This sub-set, however, consists of 1000 benign samples and 1000 malware samples. Statistics for this dataset is presented in table 7. The statistics show that the malware samples contain significantly fewer nodes and edges. This may be because benign software often com with a user interface, which adds nodes and edges, while malware usually does not. This, however, has very little affect on our results, as we only use the first 20 nodes of each sample in this experiment.

Table 7: Dataset statistics for the preliminary experiment

	Benign		Malware	
Number of samples	1000		1000	
	Nodes	Edges	Nodes	Edges
Min	0	0	1	0
Max	379986	530637	41995	52519
Sum	11590295	16111046	3249779	4179710
Mean	11590.30	16111.05	3249.78	4179.71

4.3.2 Sub-methods used

The preliminary experiments re-implements the method created by Kruegel et al. To do this, one sub-method from each of the three first parts were combined. For part 4, all four of the included classifiers were evaluated. The sub-methods used are:

Part 1: Multiple graph mode were used, due to the problems described in section 4.2.1.

Part 2: In this experiment, we only used the first 20 nodes from each sample, and only samples with 20 or more nodes were included, to make the experiment as similar to the original as possible. Additionally, samples identified as packed were removed.

Part 3: Node-based adjacency matrix was used, as this sub-methods resembles the one used in [7].

Part 4: All classifiers were used in this experiment.

4.3.3 Results

The method was evaluated using the 10-fold cross-validation method describe in section 4.4.9. Table 8 lists the classification accuracies of the method used with all four classifiers, using different percentages of the dataset as training, as well as the mean. With all classifiers except for Naive Bayes, we see that the classification accuracy increases slightly as the percentage used for training increases. That the accuracies only increases slightly indicates that all methods generalizes well, as if they did not, they would perform significantly worse with little training data. Naive Bayes is less dependent on the number of training samples, as it can learn the distribution from both few and may samples, which is why the accuracy remains more or less the same.

Table 8: Classification accuracies of the preliminary experiment, using different percentages for training, and the mean.

Classifier	Percentage training samples									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Random Forrest	0.85	0.87	0.89	0.89	0.89	0.91	0.91	**0.92**	0.90	0.89
Naive Bayes	0.72	0.73	0.72	0.72	0.72	0.72	**0.74**	0.72	0.73	0.72
Support Vector Machine	0.79	0.83	0.87	0.86	0.87	0.88	**0.89**	0.88	0.84	0.86
Multilayer perceptron	0.79	0.84	0.84	0.84	0.86	0.82	**0.88**	0.86	0.85	0.84

4.3.4 Implications

This preliminary experiment was included in the thesis to determine how well the original method, created by Kruegel et al. [7], performed with our dataset. In this experiment, we attempted to re-implement their method, and evaluate it using a modern dataset of benign software and malware. Using this method to perform binominal classification, we achieved a mean classification accuracy of 89% using the Random Forrest classifier. Kruegel et al. does not present results for malware detection in [7], but we can still compare our method to other methods. Sand achieved in [68] a classification accuracy of 98.9%, and Anderson et al. an accuracy of 96.41% in [96], but both these methods used dynamic analysis. Bragen used a method based on static analysis in [97], and achieved an accuracy of 95.58%. This shows that our method is capable of detecting malware, although not as accurately as existing methods, which indicates that we will obtain similar relative results when using the method to perform multinomial classification of malware families. It is worth noting that in this comparison, we used significantly shorter feature vectors than the other methods as the evaluated method was based on the original method [7], which may have negatively affected our results. In the final, multinomial experiments, we will be using longer feature vectors, which will make our results more comparable to those of other methods.

4.4 Experiment setup

This section describes the setup used during the experiments, and how the different sub-methods were combined. Four experiments were performed, each based on a different combination of sub-methods from part 2 of the proposed method. An overview of the experiment setup is presented in figure 18.

The different experiments are set up to evaluate different combinations of sub-methods, and to test every combination of the included sub-methods. The included sub-methods are:

Part 1

Out of the two proposed sub-methods, only multiple graph mode (MGM) were included, due to the reasons stated in section 4.2.1.

Part 2

Two sub-methods are included and combined into four different permutations:

1. No pre-processing
2. Ordering Functions (OF)
3. Limiting Nodes (LN)
4. Both Ordering Functions (OF) and Limiting Nodes (LN)

Part 3

Five sub-methods are included:

1. Node-based Adjacency Matrix with colors (NAMc)
2. Color-based Adjacency Matrix with colors (CAMc)
3. Color-based Adjacency Matrix (CAM)
4. Successor Based (SB)
5. Depth First Search colors (DFSc)

Part 4

Four classifiers were included:

1. Random Forrest (RF)
2. Naive Bayes (NB)
3. Support Vector Machine (SVM)
4. Multi-layered perceptron (MLP)

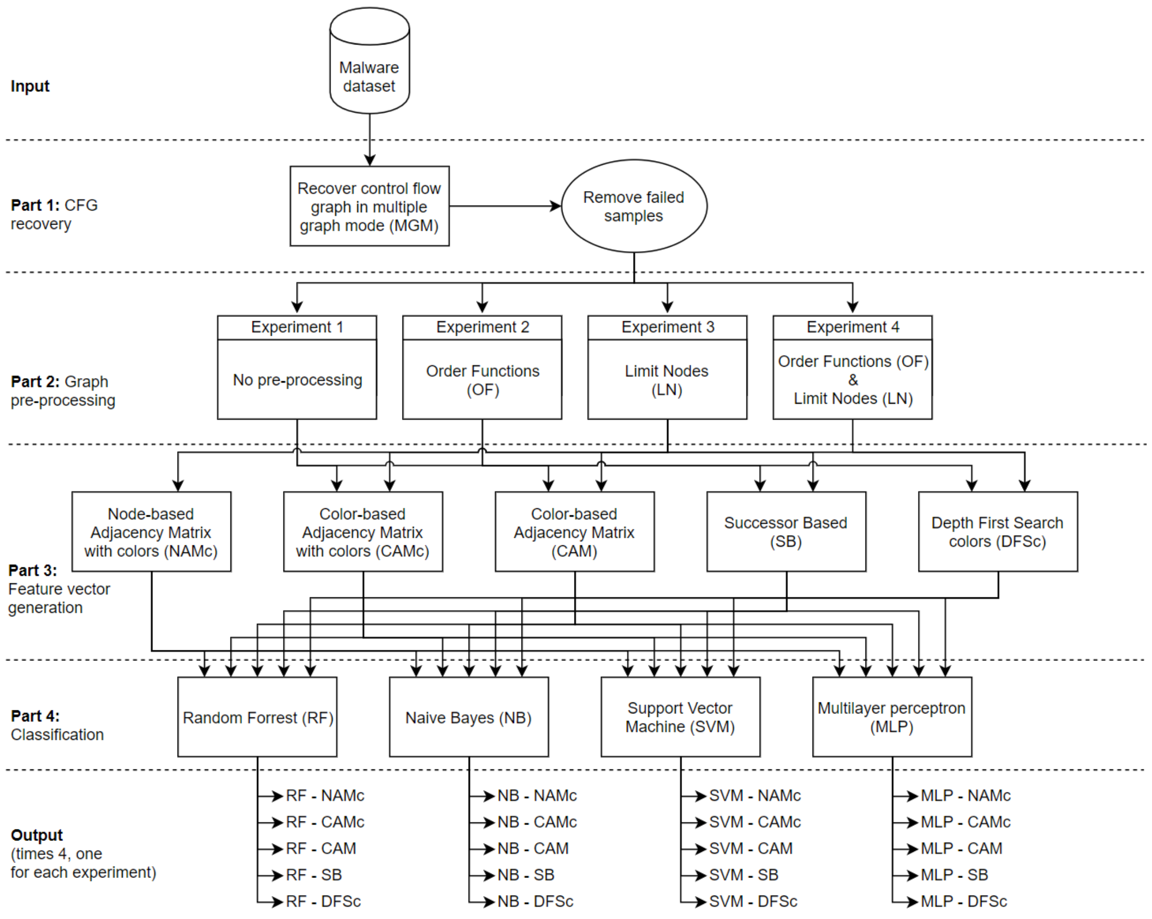


Figure 18: Experiment setup, showing the four different experiments and the sub-methods they evaluate.

Let Ξ be the set of possible implementations of the proposed method. That is, every possible permutation created by combining one sub-method from each part of the experiment. Furthermore, R, P, F, C defines the sets of sub-methods in part 1, 2, 3, and 4, respectively. Therefore:

$$\forall r \in R, \forall p \in P, \forall f \in F, \forall c \in C :$$
$$\Xi = \{r, p, f, c\}$$

This mathematical notation may at this point seem unnecessarily rigid. This is deliberate, as the number of permutations rapidly increases as new sub-methods are included, and would become disorganized with a less rigid notation. If the proposed method is further evaluated in future work, this notation will help when developing new sub-methods and comparing them to existing combinations.

The experiments, denoted E_1, E_2, E_3, E_4, evaluates different sub-sets of Ξ, all using a different p, such that:

$$(E_1, E_2, E_3, E_4) \subset \Xi$$

and $E_i \cap E_j = \emptyset$ where $\forall (i, j) \in \{1, ..., 4\}$ and $i \neq j$.

Every experiment evaluates a different sub-method from part 2 against combinations of sub-methods for the other parts. The experiments were split this way for structural reasons, not because any specific significance is place on part 2 over the other parts. As only one sub-method from part 1 is evaluated, part 2 is the first logical split in the permutations, and experiments are therefore divided using these sub-methods.

In all the experiments, the sets of included for parts 1, 3 and 4 sub-methods are defined as:

$$R_E = \{MGM\}$$

$$C_E = \{RF, NB, SVM, MLP\}$$

4.4.1 Experiment 1

The first experiment implement the method without using any pre-processing sub-methods. This serves as a baseline for comparison, to determine if pre-processing improves classification accuracy.

$$p_1 = \emptyset$$
$$F_1 = \{CAMc, CAM, SB, DFSc\}$$
$$\forall r \in R_E, \forall f \in F_1, \forall c \in C_E :$$
$$E_1 = \{r, p_1, f, c\}$$

4.4.2 Experiment 2

The second experiment adds ordering of function (OF), to compare how this sub-method affects classification accuracy.

$$p_2 = \{OF\}$$
$$F_2 = \{CAMc, CAM, SB, DFSc\}$$
$$\forall r \in R_E, \forall f \in F_2, \forall c \in C_E :$$
$$E_2 = \{r, p_2, f, c\}$$

4.4.3 Experiment 3

The third experiment limits the number of nodes (LN) used when generating the feature vectors, to determine how reducing the amount of information affects classification accuracy.

$$p_3 = \{LN\}$$
$$F_3 = \{NAMc, CAMc, CAM, SB, DFSc\}$$
$$\forall r \in R_E, \forall f \in F_3, \forall c \in C_E :$$
$$E_3 = \{r, p_3, f, c\}$$

4.4.4 Experiment 4

The final experiment combines ordering of functions (ON) and limiting nodes (LN) to determine how combining them affects classification accuracy. Note that ON is performed before LN is applied, as described in section 3.2.5.

$$p_4 = \{OF, LN\}$$
$$F_4 = \{NAMc, CAMc, CAM, SB, DFSc\}$$
$$\forall r \in R_E, \forall f \in F_4, \forall c \in C_E :$$
$$E_4 = \{r, p_4, f, c\}$$

4.4.5 Number of evaluated permutations

In total, the experiments evaluates 1 sub-method from part 1, 4 from part 2, 5 from part 4 and 4 from part 4. The number of permutations evaluated, and therefore the number of different implementations of the proposed method, was originally supposed to be:

$$1 * 4 * 5 * 4 = 80$$

However, as described in section 3.3.1, NAMc cannot be used in part 3 without LN in part 2. NAMc was therefore only used in experiment 3 and 4, where LN was used, and the number of permutations is therefore:

$$1 * 2 * (4 * 4 + 5 * 4) = 72$$

4.4.6 Minimum analysis requirements used in the experiments

Section 4.2.2 describes how minimum requirements must be set when implementing the proposed method. These requirements filters out samples where the method is not able to recover a representative control flow graph. In the experiments, a threshold of 20 nodes were used, such that any samples with less than 20 nodes were excluded. Additionally, any sample identified as packed with UPX was removed. The number of removed samples are presented in the results, in table 9.

4.4.7 Feature vector lengths

Without LN, the resulting feature vectors are too long to be used in classification. For experiment 1 and 2, where LN is not used, feature vectors are truncated after generation. For experiment 3 and 4, where LN is used, the length of the longest feature vectors is used, and shorter vectors are padded with zeroes (this only applies to CAMc and CAM, as with the others, vector lengths are dependent on node numbers).

In both cases, the mean number (rounded down to the nearest integer) is used as the length. In the first case, the mean length of the generated feature vectors is used, and all other feature vectors are truncated or padded to this length. In the second case, with LN, the mean number of nodes in all samples are used as the limit, and samples are truncated or padded with empty nodes to this length.

The mean length was chosen as the length because it better represent the dataset than any arbitrarily set number. It is worth noting that the mean is calculated for every permutation of R, F and P, resulting in different lengths.

4.4.8 Excluded sub-methods

Due to time constraints, and problems encountered during development, some of the sub-methods presented in chapter 3 were not included in the experiments. The most significant excluded method is single graph mode in part 1, which was excluded due high time usage and failure to recover the control flow graphs of a large portion of the dataset. An implementation of this sub-method using IDA Pro is described in section 4.2.1, but was left for future work due to time constraints. Comparing multiple graph mode with single graph mode, both implemented using IDA Pro, is still a relevant comparison that should be studied in future research. Other sub-methods were excluded due to time constraints, and to reduce the number of experiments.

4.4.9 Model validation

In every implementation, after generating the feature vectors, a model is created using each of the four included classifiers. The dataset is then split into two sets, a training set and a test set. The model is trained using the training set, and tested using the test set. By splitting the dataset into two sets, we ensure that the model is tested with only new samples. This tests the model's capability to classify new samples that it has not encountered before, and reduces the probability of overfitting the model. The two sets are created by randomly assigning samples from the original set into each one, until all samples have been assigned. However, this may result in an unlucky split, where some classes are overrepresented in the training set, resulting in selection bias. Each model must therefore be created, trained and tested multiple times, using different training and test sets, to get a proper evaluation of their accuracy.

These types of validation techniques are usually referred to as cross-validation techniques. The most common of these techniques are five-fold and ten-fold cross-validation, where the dataset is split into 5 and 10 parts, respectively. The model is then validated once for each part, using it as the test set while the remaining parts in each iteration make up the training sets.

In this thesis, a slightly different cross-validation technique was used to validate the models. Each created model is validated nine times, using nine different training and test sets. The key difference between this technique and the previously mentioned once is that the percentage of the dataset used in each set varies with each validation of a given model. In the first iteration, 10% of the dataset is used as the training set, and the remaining 90% is used as the test set. The sizes of training and test sets then increases and decreases, respectively, by 10% for each iteration, with the final, ninth iteration using 90% of the samples for training and 10% for testing. A new model is created, trained and validated for each iteration. The dataset is randomly shuffled at the start of each iteration.

By changing the percentage of the split with every iteration, the technique improves detection of overfitting and selection bias. In general, the classification accuracy should increase slightly with each iteration, as each new model is trained with more samples that the previous and therefore has more samples to learn from. Any significant positive divergence from this line may be caused by overfitting, and negative divergence from this line may caused by selection bias. Any overfitting will be most apparent in the later iteration, and any selection bias will be most apparent in the earlier iterations. Additionally, by calculating the mean accuracy over nine different models, we obtain a more representative view of the classifiers performance for the given input.

A drawback of this validation method is that each percentage split is only validated once. It is therefore possible to get an unlucky split between training and testing, where one or more classes are underrepresented in the training data. This will have a significant negative effect on the classification accuracy, as the model is not able to learn information about the underrepresented classes. The chance of getting an unlucky split increases as the percentage of training samples decreases. Five-fold and ten-fold cross-validation counters unlucky splits by validating the model multiple times using different training and test data, as getting multiple, unlucky splits is rather unlikely. Our model validation method can be improved by executing each iteration multiple times, randomly suffering the dataset before splitting each time. This counters unlucky splits, but significantly increases the number of evaluations, and was not included in this thesis due to time constrains, and the already large number of experiments and evaluated implementations. Further analysis of this model validation method is left for future work.

4.4.10 Software used

The experiments were implemented using the Python programming language [98]. The software used when implementing control flow graph recovery in multiple graph mode is described in table 4.2.1. IDA Pro, The Interactive Disassembler [62], was used recover the graphs. Two Python 2.7

scripts were used, one to interact with IDA Pro using the IDAPython API [99], and one helper script to execute the first script on all samples and handled parallelization. These scripts are presented in appendix A. For security reasons, control flow graph recovery was performed in a virtual machine with guest isolation enabled, to reduce the risk of accidental infection.

The two other scripts are implemented using Python 3.7. the first implements the part 2 and 3 sub-methods, except for OF, which requires information from IDA Pro, and was therefore implemented in the IDAPython script. This script is presented in appendix B, with the color groups presented in appendix C. The final script implements the classification sub-methods from part 4, using the scikit-learn framework, and is presented in appendix D.

5 Results

This chapter presents and explains the results of the experiments. It is divided into four sections, one for each part of the proposed method. Each section presents the results of every evaluated implementation of the method for that part of the method. The first section presents the results of control flow recovery, as well as a breakdown of the how the different classes were affected by the removal in this part. The second presents the results of graphs based pre-processing, and the number of nodes and edges in the different classes. The third presents the lengths of the generated feature vectors. The final section presents the classification results, including boxplots for comparison.

5.1 Control flow graph recovery results

The first part of the method involves recovering the control flow graph from malware samples. In experiments, only one sub-method was included for this part: multiple-graph mode (MGM). The results are presented in table 9. The table includes the original number of samples in the dataset, the number of samples that were removed and the number of samples that used in the experiments. Three reasons are given for the removed samples. Two of these, removed packed and remove <20 nodes, are described in section 4.4.6. The third, failed analysis, was included to capture samples where the sub-method was unable to recover the control flow graph or necessary metadata. These cases are described below. It is worth noting that a given sample could have been removed for more than one of these reasons, which is why they add up to more than the number listed as total removed. For instance, several samples detected as packed also resulted in graphs with less than 20 nodes.

Table 9: Experiment results for part 1 of the proposed method. Including normal recovery (experiment 1 and 3), and recovery used with ordered functions pre-processing (experiment 2 and 4).

	Normal	**Ordered functions**
Initial number of samples	9823	9823
Failed analysis	0	2
Removed packed	915	915
Removed <20 nodes	4368	4368
Total removed	4476	4478
Total included samples	5347	5345

Table 9 shows two different results: normal and Ordered functions. The ladder is a part 2 sub-method, but is included here has it requires information only available during part 1., In both cases, we started of with 9823 malware samples, of which 915 were identified as packed, and 4368 resulted in graphs with less than 20 nodes. As described in section 4.2.2, a threshold of 20 nodes would exclude about 45% of our dataset. This corresponds well with the results we got. Some additional samples, identified as packed, were also removed. However, as most of these were also removed for having too few nodes, they did not have a significant effect on the resulting number. As a result, we ended up with 5347 and 5345 samples, which is close to our original estimate of about 55%.

The one difference between the two results is that we excluded samples when Ordered functions was applied. This sub-method relies on the entry point in the PE header. In the two samples were recovery failed, this entry point was obfuscated, resulting in Ordered functions failing to pre-process them. The samples were therefore removed in the experiments that included this sub-method, and the slightly altered result of part 1 for experiment 2 and 4 are included in table 9 for comparison. These differences has minimal effect on the later results of the experiments with regards to comparison, as only two samples were removed.

The different classes in the dataset were affected differently by the removal of samples. The original number of samples within each class, and the resulting numbers, for both normal and Ordered functions, are presented in figure 19. Compared to the original distribution of samples, where nine classes contained 1000 samples and one contained 823, this new distributions are rather uneven. Three outliers can be seen in the chart, one above most of the others, and two below. In the outlier above, Renos, very few samples were removed. This indicates that very few of these samples were identified as packed or were obfuscated in some other way. The opposite is the case for the two bottom outliers, Hupigon and Onlinegames, where the method were only able to recover the control flow graph from some of the samples. One special case is Vundo, where very few samples were removed, but since we started out with fewer samples in this class, it is not considered an outliner in the resulting distribution. The remaining six families, Agent, Obfuscator, Small, Vb, Vbinject and Zlob, had a relatively similar reduction, leaving us with between 40% and 70% of the original samples.

No further adjustments were made to the distribution of samples between the classes. As a result, the differences with regards to number of samples, especially between the outliers, may affect the classification results. However, as we started out with such a large number of samples, the removal of about 80% of the samples still laves us with enough for classification. We therefore did not exclude any of the classes from the remaining experiments.

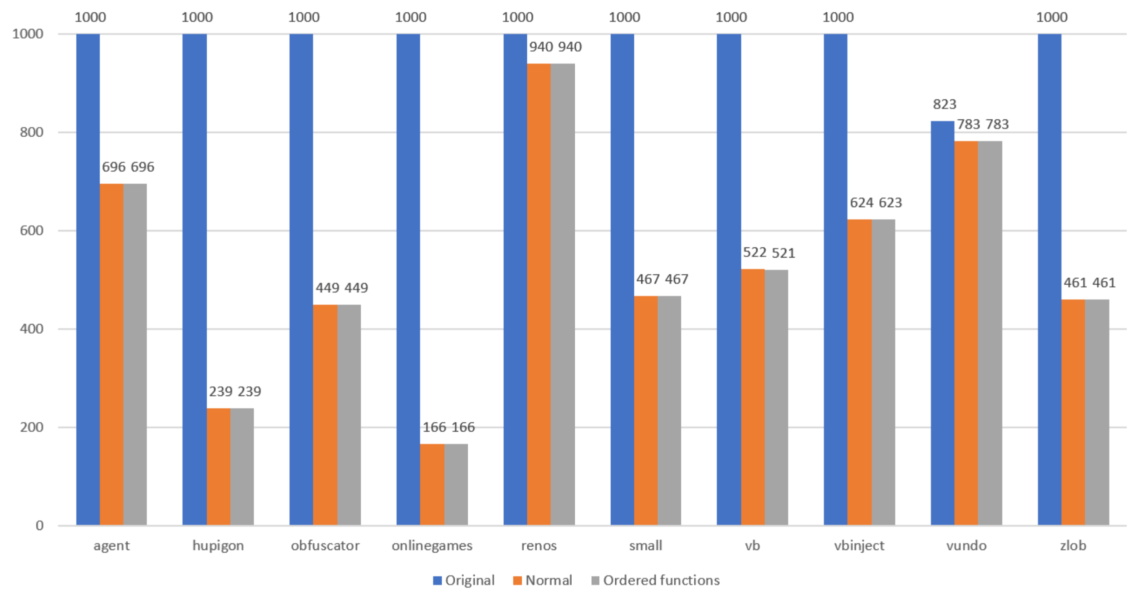


Figure 19: Barchart of the number of samples in each class after removal in part 1. Normal is simply multiple graphs mode with IDA Pro, while Ordered functions included this part 2 method.

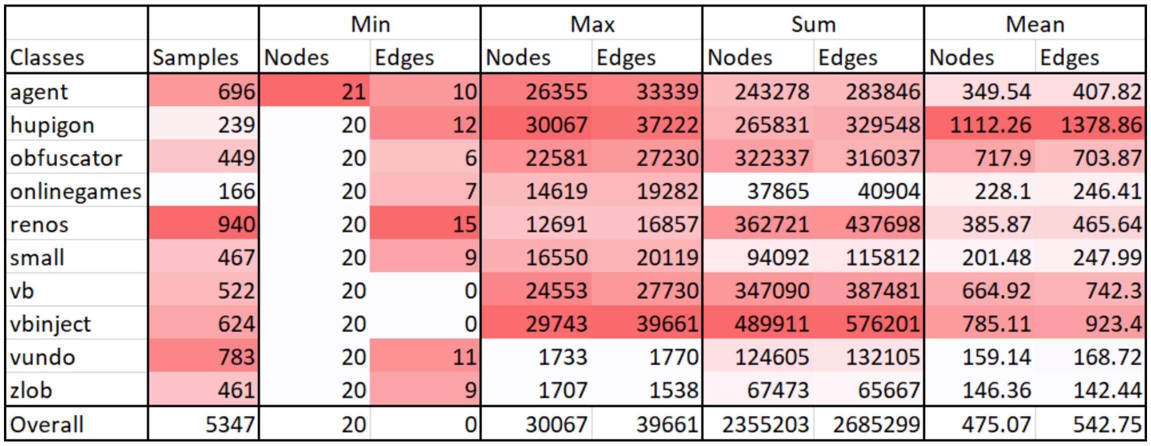
5.2 Graph based pre-processing results

The second part of the experiment involves pre-processing of graphs. Four different sub-methods were used for this part, one for each experiment. The results for part 2 of experiment 1, 2, 3 and 4 are presented in figures 20, 21, 22 and 23, respectively. Each figure presents the results after removal of insufficient samples in part 1. The number of samples within each class is included to highlight how the different classes were affected differently by the removal in part 1. For each class, we include the minimum, maximum, sum and mean number of nodes and edges, as well the overall results for all included samples. A heatmap has been applied to each column, to highlight extreme values. Smaller values are marked in white, while higher values are red.

The most significant differences in the results are between the experiments that included Limit nodes, and the once that did not. Ordered functions, used in experiment 2 and 4, did not have any significant effect on the number of nodes and edges. This was expected, as this pre-processing sub-method only reordered the nodes, without removing any.

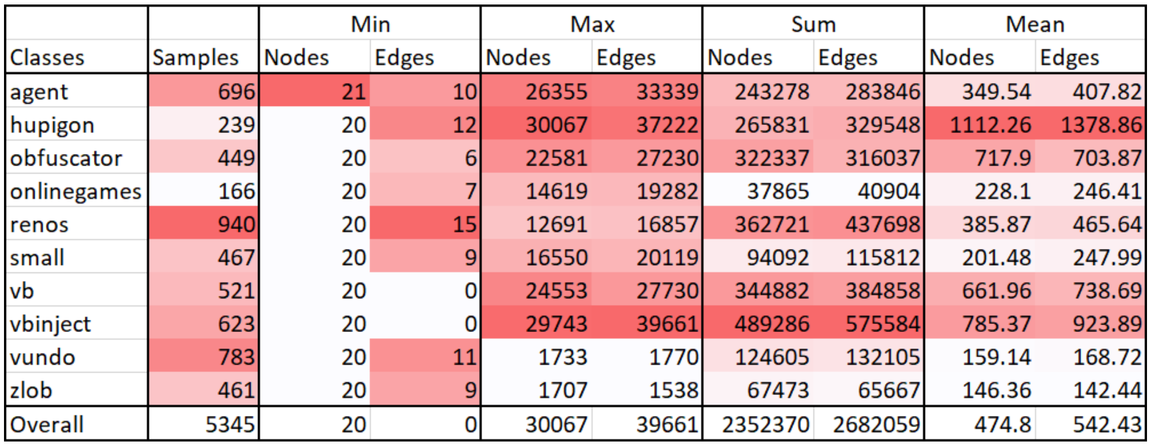
Figures 20 and 21 show the results of pre-processing in experiments 1 and 2, respectively. Experiment 1 served as the baseline for the experiments, and therefore included no pre-processing sub-methods. Experiment 2 used only Ordered functions to pre-process the graphs. As the results are at this point in the experiments almost identical, we will describe them as one in the rest of this section. Firstly, the minimum number of nodes is 20 for most classes, as this was the threshold.

For the first class, Agent, the minimum number of nodes is 21, indicating that, while there were samples with less than 20 nodes, no sample in that class had exactly 20 nodes. The minimum number of edges show that for both Vb and Vbinject, some samples contained no edges. The number of edges required to link every node to at least one other node in the graph is $\lceil N/2 \rceil$, where N is the number of nodes in the graph. Therefore, 4 classes, Agent, Obfuscator, Onlinegames and Zlob, contained fewer edges than required to link every node to at least one other nodes, given the minimum number of nodes in the classes.



		Min		Max		Sum		Mean	
Classes	Samples	Nodes	Edges	Nodes	Edges	Nodes	Edges	Nodes	Edges
agent	696	21	10	26355	33339	243278	283846	349.54	407.82
hupigon	239	20	12	30067	37222	265831	329548	1112.26	1378.86
obfuscator	449	20	6	22581	27230	322337	316037	717.9	703.87
onlinegames	166	20	7	14619	19282	37865	40904	228.1	246.41
renos	940	20	15	12691	16857	362721	437698	385.87	465.64
small	467	20	9	16550	20119	94092	115812	201.48	247.99
vb	522	20	0	24553	27730	347090	387481	664.92	742.3
vbinject	624	20	0	29743	39661	489911	576201	785.11	923.4
vundo	783	20	11	1733	1770	124605	132105	159.14	168.72
zlob	461	20	9	1707	1538	67473	65667	146.36	142.44
Overall	5347	20	0	30067	39661	2355203	2685299	475.07	542.75

Figure 20: Number of samples, nodes and edges for every class in experiment 1. No pre-processing was applied in this experiment. A heatmap is applied to each column to highlight high (red) and low (white) values.

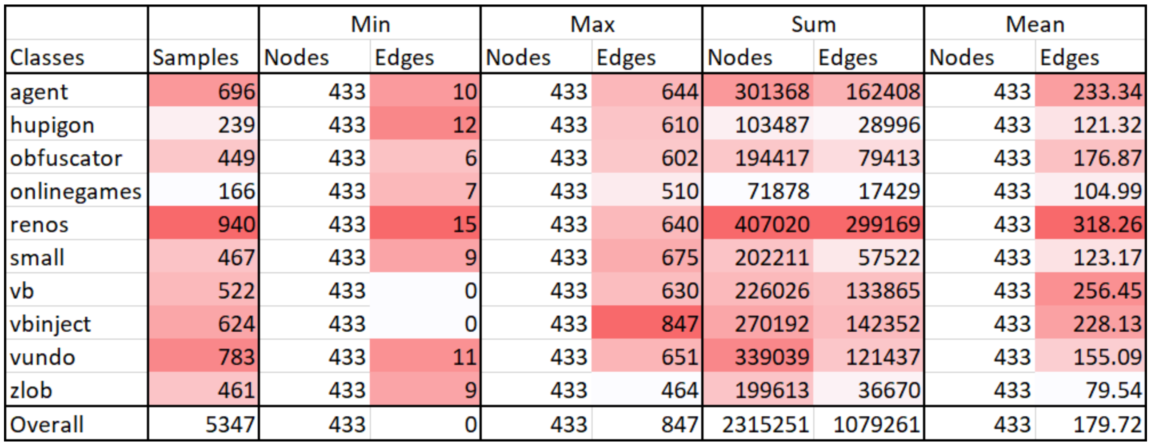


		Min		Max		Sum		Mean	
Classes	Samples	Nodes	Edges	Nodes	Edges	Nodes	Edges	Nodes	Edges
agent	696	21	10	26355	33339	243278	283846	349.54	407.82
hupigon	239	20	12	30067	37222	265831	329548	1112.26	1378.86
obfuscator	449	20	6	22581	27230	322337	316037	717.9	703.87
onlinegames	166	20	7	14619	19282	37865	40904	228.1	246.41
renos	940	20	15	12691	16857	362721	437698	385.87	465.64
small	467	20	9	16550	20119	94092	115812	201.48	247.99
vb	521	20	0	24553	27730	344882	384858	661.96	738.69
vbinject	623	20	0	29743	39661	489286	575584	785.37	923.89
vundo	783	20	11	1733	1770	124605	132105	159.14	168.72
zlob	461	20	9	1707	1538	67473	65667	146.36	142.44
Overall	5345	20	0	30067	39661	2352370	2682059	474.8	542.43

Figure 21: Number of samples, nodes and edges for every class in experiment 2 after pre-processing with OF. A heatmap is applied to each column to highlight high (red) and low (white) values.

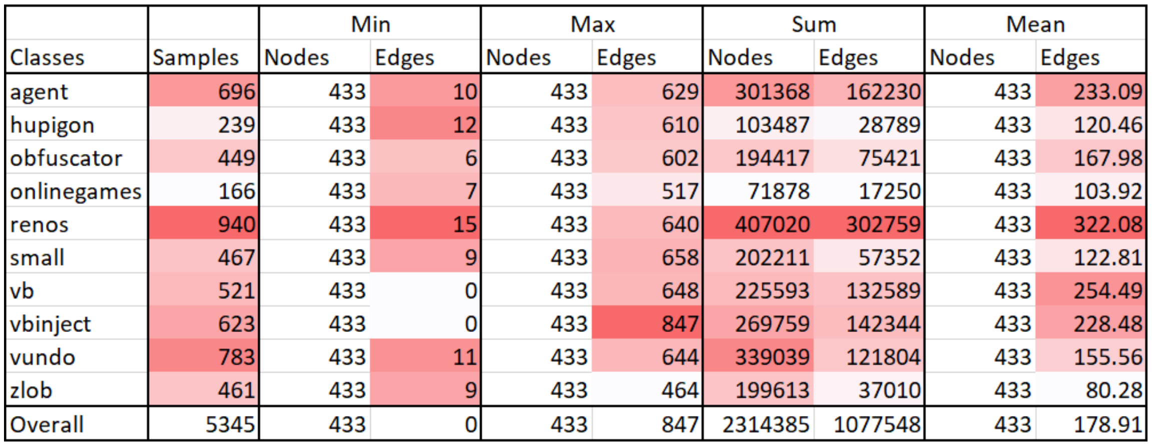
Secondly, the maximum number of nodes and edges are shown in the next two columns. Here, the heatmap is highly skewed by Vundo and Zlob, as the largest samples in both of these classes had significantly fewer nodes and edges than in the rest. The larges sample, with regards to number of nodes, belonged to the Hupigon class with 30067 nodes, and the largest with regards to edges belonged to Vbinject with 39661 edges. Thirdly, the sum of nodes and edges included. Here we see that, in all cases, the numbers of nodes and edges are very similar. Onlinegames had the fewest nodes and edges, with 37865 and 40904, while Vbinject had the most, with 489911 and 576201. Finally, the the mean number of nodes and edges are presented. Again, we see a slight correlation between the numbers for nodes and edges. These numbers factors out the number of samples within each class, enabling a fair comparison between classes with considering the number of samples. For example, while Hupingon had the greatest mean number of nodes and edges, at 1112.26 and 1378.86, its sums were only about average, as there were very few samples within this class. Vundo and Zlob had the lowest mean values, which correlates with their maximum values.

Experiments 3 and 4 are shown in figures 20 and 21, respectively. Both experiments used Limit nodes to pre-process the graphs, reducing the number of nodes in each graph to 433, which was the mean number of nodes in the dataset. Additionally, shorter graphs were padded with empty nodes to a length of 433, and any edges to or from removed nodes were also removed. Experiment 4 also included Ordered functions. As with the first two experiments, the results in these two are also almost identical, and will be presented as one for the rest of the section. As Limit nodes were applied, all graphs had a length of 433 nodes, which is shown in the figures in the nodes columns. The minimum number of edges remain the same as in the first two experiments.



		Min		Max		Sum		Mean	
Classes	Samples	Nodes	Edges	Nodes	Edges	Nodes	Edges	Nodes	Edges
agent	696	433	10	433	644	301368	162408	433	233.34
hupigon	239	433	12	433	610	103487	28996	433	121.32
obfuscator	449	433	6	433	602	194417	79413	433	176.87
onlinegames	166	433	7	433	510	71878	17429	433	104.99
renos	940	433	15	433	640	407020	299169	433	318.26
small	467	433	9	433	675	202211	57522	433	123.17
vb	522	433	0	433	630	226026	133865	433	256.45
vbinject	624	433	0	433	847	270192	142352	433	228.13
vundo	783	433	11	433	651	339039	121437	433	155.09
zlob	461	433	9	433	464	199613	36670	433	79.54
Overall	5347	433	0	433	847	2315251	1079261	433	179.72

Figure 22: Number of samples, nodes and edges for every class in experiment 3 after pre-processing with LN. A heatmap is applied to each column to highlight high (red) and low (white) values.



		Min		Max		Sum		Mean	
Classes	Samples	Nodes	Edges	Nodes	Edges	Nodes	Edges	Nodes	Edges
agent	696	433	10	433	629	301368	162230	433	233.09
hupigon	239	433	12	433	610	103487	28789	433	120.46
obfuscator	449	433	6	433	602	194417	75421	433	167.98
onlinegames	166	433	7	433	517	71878	17250	433	103.92
renos	940	433	15	433	640	407020	302759	433	322.08
small	467	433	9	433	658	202211	57352	433	122.81
vb	521	433	0	433	648	225593	132589	433	254.49
vbinject	623	433	0	433	847	269759	142344	433	228.48
vundo	783	433	11	433	644	339039	121804	433	155.56
zlob	461	433	9	433	464	199613	37010	433	80.28
Overall	5345	433	0	433	847	2314385	1077548	433	178.91

Figure 23: Number of samples, nodes and edges for every class in experiment 4 after pre-processing with OF and LN. A heatmap is applied to each column to highlight high (red) and low (white) values.

The first difference of interest, comes in the maximum number of edges, were we see a significant reduction. This was expected, as we only included edges to and from the first 433 nodes, removing any other edges. The second difference of interest comes in the sum columns. The total number of edges has decreased for all classes, which follows from the decrease in maximum number of edges. The total number of nodes have increased or decreased, depending on the classes mean number of nodes in experiment 1 and 2, compared to the overall mean. Classes with higher mean number of nodes in the first two experiments, Hupingon, Obfuscator, Vb and Vbinject, have decreased in total number of nodes, while the classes with lower mean numbers, Agent, Onlinegames, Renos, Small, Vundo and Zlob, have increased. Effectively, the total number of nodes in each class has become equal to the number of samples in that class, multiplied by 433. Finally, the figures show a slight correlation between the sum of edges, and the mean number of edges, which does not appear in the first two experiments. This is somewhat unexpected, as the sum is dependent on the number of samples, while the mean value removes this factor. For example, it is expected that Renos, with the greatest number of samples, 940, also should have the greatest total number of edges, 299169. However, this does not imply that it also has the highest mean number of edges.

5.3 Feature vector generation results

Part 3 of the method generates feature vectors from the control flow graphs. Four sub-methods were evaluated in experiment 1 and 2, and five in experiment 3 and 4. The results for experiment 1, 2, 3 and 4 are presented in tables 10, 11, 12 and 13, respectively. Each table lists the minimum, maximum and mean feature vector length for each sub-method.

As with the previous part, there are minimal differences when comparing experiment 1 to 2 and 3 to 4, as also these results are based on the number of nodes and not the ordering. That is, the only observable difference from applying Ordered functions comes from the two removed samples. More significant differences can be observed when comparing experiments 1 and 2 to experiments 3 and 4.

Table 10: Experiment 1 feature vector lengths.

Method	**Min length**	**Max length**	**Mean length**
Node-based AM + color	N/A	N/A	N/A
Color-based AM + color	2	274052	6635.57
Color-based AM	1	273529	6581.68
Successor based	100	150335	2202.36
Depth first search colors	20	30067	440.47

Table 11: Experiment 2 feature vector lengths.

Method	**Min length**	**Max length**	**Mean length**
Node-based AM + color	N/A	N/A	N/A
Color-based AM + color	2	274052	6631.81
Color-based AM	1	273529	6577.95
Successor based	100	150335	2200.53
Depth first search colors	20	30067	440.11

Tables 10 and 11 shows statistics for experiment 1 and 2. Node-based adjacency matrix with color (NAMc) was not used in these, as it generates feature vectors with a length equal to $k^2 + k$, where k is the number of nodes. In the previous section, in figures 20 and 21, we saw that the largest graphs had 30067 nodes. This would have resulted in a feature vector length of 904054556, which exceeds the practical limits of most computers.

For all sub-methods, we see that the length of the generated feature vectors are dependent on the number of nodes. In tables 10 and 11, the minimum length for Color-based adjacency matrix with color (CAMc) and Color-based adjacency matrix without color (CAM) is 2 and 1, respectively. These minimum values represent feature vectors of graphs where every node had the same color, resulting in a 1*1 adjacency matrix. The maximum values for CAMc and CAM are a result of the graphs with the greatest number of different colors. The lengths of these sub-methods are equal to:

$$CAMc : c^2 + c$$

$$CAM : c^2$$

We can therefore calculate the minimum, maximum and mean number of different colors per sample in the dataset, using the numbers in tables 10 and 11:

$$c_{min} = (c_{min}^2 + c_{min}) - (c_{min}^2) = 2 - 1 = \mathbf{1}$$

$$c_{max} = (c_{max}^2 + c_{max}) - (c_{max}^2) = 274052 - 273529 = \mathbf{523}$$

$$Experiment1 : c_{mean} = (c_{mean}^2 + c_{mean}) - (c_{mean}^2) = 6635.57 - 6581.68 = \mathbf{53.89}$$

$$Experiment2 : c_{mean} = (c_{mean}^2 + c_{mean}) - (c_{mean}^2) = 6631.81 - 6577.95 = \mathbf{53.86}$$

Finally, the mean lengths for CAMc and CAM were 6635.57 and 6581.68 for experiment 1 in table 10, and 6631.81 and 6577.95 for experiment 2 in table 11.

The last two sub-methods used in experiment 1 and 2 were Successor based (SB) and Depth first search colors (DFSc). Tables 10 and 11 show that SB, in both experiment 1 and 2, resulted in a minimum length of 100 and a maximum length of 150335. The mean lengths were, for experiment 1 in table 10, 2202.36, and for experiment 2 in 11, 2200.53. For DFSc, we see a minimum length of 20, maximum length of 30067, and mean lengths of 440.47 in experiment 1 and 440.11 in experiment 2.

The lengths of feature vectors generated by DFSc is equal to the number of nodes in the graphs. We can therefor see from tables 10 and 11, that the smallest graphs used in these experiments contained 20 nodes, which was the minimum number of nodes required for a graph to be included, and that the largest graphs contained 30067 nodes. These numbers are equal to the minimum and maximum number of nodes shown in figures 20 and 21 in the part 2 results.

Tables 12 and 13 shows statistics for experiment 3 and 4. In these experiments, we applied Limit nodes to reduce and normalize the number of nodes in each sample. The mean number of nodes per sample in the dataset rounded down to the nearest integer, 433, was used as the number of nodes to included from each sample. From each sample, we selected the first 433 nodes. Shorter graphs were padded with empty nodes, containing no instructions and therefore a color value of 0, to a length of 433. As a result, all graphs in these experiments had a length of 433. Any edges to or from nodes that were removed were also removed.

Table 12: Experiment 3 feature vector lengths.

Method	**Min length**	**Max length**	**Mean length**
Node-based AM + color	187922	187922	187922.0
Color-based AM + color	6	42642	3114.93
Color-based AM	4	42436	3069.99
Successor based	2165	2165	2165.0
Depth first search colors	433	433	433.0

Table 13: Experiment 4 feature vector lengths.

Method	Min length	Max length	Mean length
Node-based AM + color	187922	187922	187922.0
Color-based AM + color	6	35156	2743.04
Color-based AM	4	34969	2699.64
Successor based	2165	2165	2165.0
Depth first search colors	433	433	433.0

This reduction in nodes enabled us to include NAMc in these experiments. The lengths of feature vectors generated by NAMc are only dependent on the number of nodes in the original graphs. Therefore, as we see in tables 12 and 13, all feature vectors generated by NAMc, in both experiment 3 and 4, were of length 187922.

In table 12 we see that CAMc resulted in a minimum length of 6, maximum length of 42642, and mean length of 3114.93, and that CAM resulted in a minimum length of 4 maximum length of 42436, and mean length of 3069.99, for experiment 3. From these values, we can calculate the minimum, maximum and mean number of different colors per sample in the dataset, after applying Limit nodes in experiment 3:

$$c_{min} = (c_{min}^2 + c_{min}) - (c_{min}^2) = 6 - 4 = \mathbf{2}$$
$$c_{max} = (c_{max}^2 + c_{max}) - (c_{max}^2) = 42642 - 42436 = \mathbf{206}$$
$$c_{mean} = (c_{mean}^2 + c_{mean}) - (c_{mean}^2) = 3114.93 - 3069.99 = \mathbf{44.94}$$

For experiment 4, table 13 show that CAMc resulted in a minimum length of 6, maximum length of 35156, and mean length of 2743.04, and that CAM resulted in a minimum length of 4, maximum length of 34969, and mean length of 2699.64. The minimum, maximum and mean number of colors can then be calculated as:

$$c_{min} = (c_{min}^2 + c_{min}) - (c_{min}^2) = 6 - 4 = \mathbf{2}$$
$$c_{max} = (c_{max}^2 + c_{max}) - (c_{max}^2) = 35156 - 34969 = \mathbf{187}$$
$$c_{mean} = (c_{mean}^2 + c_{mean}) - (c_{mean}^2) = 2743.04 - 2699.64 = \mathbf{43.40}$$

The maximum number of different colors decreased from experiment 3 to experiment 4, indicating that one of the samples that failed analysis when Ordered functions was applied contained the greatest number of different colors after applying Limit nodes.

Finally, SB and DFSc are dependent on the number of nodes. Therefore, in tables 12 and 13, we see that all feature vectors generated by SB in experiments 3 and 4 had a length of 2165, which is equal to $5 * 433$. We also see that all vectors generated by DFSc had a length of 433.

5.4 Classification results

Part 4 of the experiments uses the feature vectors generated in part 3 and performs multinomial malware classification. The results are presented in tables 14, 15, 16 and 17. In each experiment, every set of feature vectors generated in part 3 was combined with every classifier, and every combination was evaluated using the cross-validation method described in section 4.4.9. For every combination, the classification accuracy of all nine models are listed, ordered by the percentage of samples used for training, as well as the mean accuracy. The greatest value(s) of each row has been highlighted in bold. Additionally, boxplots has been included to highlight classification accuracy distributions. These are described, and the results are discussed at the end of this section.

The following four pages presents the classification results of each experiment.

Table 14 presents the resulting classification accuracies of experiment 1. This experiment served as the baseline, using no pre-processing sub-methods. We see that, for Random Forrest (RF), our method achieved mean classification accuracies of 57%, 57%. 71% and 70%, with highest accuracies of 63%, 65%, 75% and 76%, for CAMc, CAM, SB and DFSc, respectively. Secondly, when combined with Naive Bayes (NB), the method achieved mean accuracies of 35%, 40%, 31% and 25%, and highest accuracies of 40%, 43%, 36% and 26%. Thirdly, for Support Vector Machine (SVM), the method achieved mean accuracies of 47%, 19%, 54% and 52%, and highest accuracies of 54%, 21%, 62% and 59%. Finally, with Multilayer perceptron (MLP), the method achieved mean accuracies of 48%, 54%, 58% and 51%, and highest accuracies of 54%, 60%, 67% and 57%.

Table 14: Experiment 1 classification accuracy results.

	Random Forrest									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color										
Color-based AM + color	0.45	0.50	0.55	0.57	0.61	0.58	0.60	0.62	**0.63**	0.57
Color-based AM	0.46	0.51	0.56	0.57	0.59	0.60	0.60	0.60	**0.65**	0.57
Successor based	0.63	0.69	0.69	0.72	0.72	0.73	0.74	**0.75**	0.73	0.71
DFS colors	0.61	0.65	0.70	0.72	0.71	0.71	0.73	0.74	**0.76**	0.70

	Naive Bayes									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color										
Color-based AM + color	0.31	0.35	0.36	0.36	0.35	0.36	0.35	**0.40**	0.36	0.35
Color-based AM	0.32	0.35	0.39	0.39	0.42	0.41	0.42	**0.43**	**0.43**	0.40
Successor based	0.26	0.34	0.31	0.28	0.32	0.33	0.31	0.32	**0.36**	0.31
DFS colors	0.25	**0.26**	0.24	0.25	**0.26**	0.25	0.25	0.24	0.23	0.25

	Support Vector Machine									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color										
Color-based AM + color	0.33	0.40	0.46	0.46	0.49	0.50	0.51	0.53	**0.54**	0.47
Color-based AM	0.18	0.18	0.19	0.19	0.19	0.19	0.20	**0.21**	**0.21**	0.19
Successor based	0.39	0.48	0.51	0.55	0.58	0.57	**0.62**	0.59	0.61	0.54
DFS colors	0.38	0.47	0.49	0.52	0.55	0.56	0.58	0.57	**0.59**	0.52

	Multilayer perceptron									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color										
Color-based AM + color	0.39	0.41	0.44	0.46	0.52	0.52	0.50	0.50	**0.54**	0.48
Color-based AM	0.40	0.49	0.53	0.53	0.55	0.58	**0.60**	0.59	0.59	0.54
Successor based	0.45	0.49	0.56	0.56	0.59	0.58	0.64	0.64	**0.67**	0.58
DFS colors	0.41	0.45	0.49	0.49	0.54	0.56	0.54	0.55	**0.57**	0.51

Table 15 presents the resulting classification accuracies of experiment 2. In this experiment, Ordered functions were applied. The table shows that, with RF, our method achieved mean classification accuracies of 58%, 57%, 74% and 72%, and highest accuracies of 66%, 63%, 79% and 77%, for CAMc, CAM, SB and DFSc, respectively. With NB, the method achieved mean accuracies of 35%, 40%, 32% and 22%, and highest accuracies of 37%, 44%, 34% and 23%. With SVM, the method achieved mean accuracies of 48%, 19%, 55% and 52%, and highest accuracies of 58%, 20%, 60% and 61%. Finally, with MLP, the method achieved mean accuracies of 47%, 55%, 57% and 52%, and highest accuracies of 53%, 61%, 63% and 59%.

Table 15: Experiment 2 classification accuracy results.

	Random Forrest									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color										
Color-based AM + color	0.47	0.52	0.55	0.57	0.60	0.61	0.61	0.63	**0.66**	0.58
Color-based AM	0.46	0.52	0.54	0.57	0.59	0.62	0.62	0.60	**0.63**	0.57
Successor based	0.66	0.70	0.70	0.74	0.74	0.77	0.77	0.78	**0.79**	0.74
DFS colors	0.61	0.67	0.71	0.71	0.74	0.74	**0.77**	0.73	0.75	0.72

	Naive Bayes									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color										
Color-based AM + color	0.33	0.35	**0.37**	0.34	0.35	0.35	**0.37**	0.35	**0.37**	0.35
Color-based AM	0.34	0.36	0.41	0.40	0.41	0.41	0.40	0.41	**0.44**	0.40
Successor based	0.32	0.31	0.33	0.33	**0.34**	0.33	0.31	0.32	0.32	0.32
DFS colors	0.19	0.23	0.23	0.22	0.22	0.21	0.24	0.21	**0.23**	0.22

	Support Vector Machine									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color										
Color-based AM + color	0.33	0.41	0.45	0.47	0.50	0.48	0.53	0.54	**0.58**	0.48
Color-based AM	0.18	0.19	0.18	**0.20**	0.19	**0.20**	0.19	0.19	0.18	0.19
Successor based	0.43	0.49	0.54	0.55	0.57	0.59	**0.60**	0.59	**0.60**	0.55
DFS colors	0.39	0.48	0.48	0.52	0.53	0.54	0.57	0.59	**0.61**	0.52

	Multilayer perceptron									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color										
Color-based AM + color	0.37	0.42	0.44	0.49	0.50	0.51	0.51	**0.53**	0.49	0.47
Color-based AM	0.41	0.48	0.52	0.55	0.56	0.59	0.60	0.59	**0.61**	0.55
Successor based	0.46	0.50	0.54	0.59	0.58	**0.63**	0.59	**0.63**	0.62	0.57
DFS colors	0.40	0.46	0.51	0.49	0.54	0.57	0.56	0.57	**0.59**	0.52

Table 16 presents the resulting classification accuracies of experiment 3. This experiment used Limit nodes for pre-processing, reducing all graphs to a length of 433 nodes. This enabled us to also evaluate Node-based adjacency matrix with colors (NAMCc). The table shows that, when using RF, our method achieved mean classification accuracies of 69%, 57%, 55%, 71% and 70%, and highest accuracies of 75%, 62%, 59%, 75% and 74%, for NAMc, CAMc, CAM, SB and DFSc, respectively. With NB, the method achieved mean accuracies of 24%, 39%, 44%, 32% and 24%, and highest accuracies of 28%, 42%, 46%, 34% and 26%. With SVM, the method achieved mean accuracies of 53%, 47%, 20%, 55% and 52, and highest accuracies of 60%, 55%, 21%, 62% and 60%. Finally, with MLP, the method achieved mean accuracies of 53%, 49%, 54%, 55% and 50%, and highest accuracies of 65%, 56%, 60%, 63% and 59%.

Table 16: Experiment 3 classification accuracy results.

	Random Forrest									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color	0.61	0.66	0.68	0.68	0.70	0.72	0.70	**0.75**	0.71	0.69
Color-based AM + color	0.46	0.52	0.55	0.58	0.57	0.60	0.59	**0.62**	0.61	0.57
Color-based AM	0.44	0.50	0.52	0.57	0.57	0.58	**0.59**	0.57	0.58	0.55
Successor based	0.62	0.66	0.70	0.72	0.73	0.73	**0.75**	**0.75**	**0.75**	0.71
DFS colors	0.61	0.66	0.69	0.71	0.71	0.73	**0.74**	**0.74**	0.73	0.70
	Naive Bayes									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color	0.21	0.25	0.22	0.25	0.25	0.25	0.25	0.21	**0.28**	0.24
Color-based AM + color	0.34	0.36	0.38	0.40	0.41	0.39	0.39	**0.42**	**0.42**	0.39
Color-based AM	0.39	0.43	0.41	0.44	0.44	**0.46**	**0.46**	0.45	0.45	0.44
Successor based	**0.34**	0.31	**0.34**	**0.34**	0.31	0.33	0.32	0.33	0.31	0.32
DFS colors	0.21	0.25	0.24	0.21	0.22	**0.26**	0.25	0.22	0.24	0.24
	Support Vector Machine									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color	0.39	0.48	0.48	0.53	0.56	0.56	0.58	0.58	**0.60**	0.53
Color-based AM + color	0.34	0.40	0.44	0.49	0.48	0.52	0.53	**0.55**	0.51	0.47
Color-based AM	0.19	0.19	0.19	0.19	0.20	**0.21**	**0.21**	**0.21**	**0.21**	0.20
Successor based	0.40	0.50	0.53	0.57	0.56	0.60	0.60	**0.62**	0.61	0.55
DFS colors	0.35	0.45	0.49	0.53	0.54	0.56	0.58	**0.60**	0.58	0.52
	Multilayer perceptron									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color	0.37	0.48	0.50	0.54	0.54	0.57	0.57	0.57	**0.65**	0.53
Color-based AM + color	0.38	0.44	0.47	0.48	0.51	0.50	**0.56**	0.51	0.55	0.49
Color-based AM	0.43	0.48	0.50	0.54	0.56	0.57	0.59	**0.60**	0.59	0.54
Successor based	0.43	0.51	0.51	0.52	0.57	0.58	0.58	**0.63**	0.62	0.55
DFS colors	0.37	0.47	0.44	0.52	0.51	0.51	0.57	0.50	**0.59**	0.50

Finally, table 17 presents the resulting classification accuracies of experiment 4. In this experiment, both Ordered functions and Limit nodes were applied. This experiment included NAMc, as with the previous one. The table shows that, when using RF, our method achieved mean classification accuracies of 70%, 57%, 56%, 73% and 72, and highest accuracies of 76%, 62%, 61%, 78% and 77%, for NAMc, CAMc, CAM, SB and DFSc, respectively. With NB, the method achieved mean accuracies of 22%, 38%, 45%, 34% and 21%, and highest accuracies of 25%, 41%, 49%, 35% and 23%. With SVM, the method achieved mean accuracies of 53%, 49%, 20%, 56% and 52%, and highest accuracies of 62%, 56%, 21%, 66% and 61%. Finally, with MLP, the method achieved mean accuracies of 54%, 50%, 54%, 56% and 52%, and highest accuracies of 62%, 55%, 60%, 62% and 58%.

Table 17: Experiment 4 classification accuracy results.

	Random Forrest									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color	0.61	0.66	0.69	0.70	0.70	0.74	**0.76**	0.73	0.75	0.70
Color-based AM + color	0.47	0.52	0.57	0.57	0.59	0.59	0.60	0.61	**0.62**	0.57
Color-based AM	0.47	0.52	0.55	0.55	0.57	0.58	0.59	**0.61**	0.59	0.56
Successor based	0.65	0.70	0.70	0.73	0.74	0.76	0.75	**0.78**	0.77	0.73
DFS colors	0.62	0.67	0.70	0.75	0.74	0.75	0.76	0.75	**0.77**	0.72
	Naive Bayes									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color	**0.25**	0.23	0.23	0.22	0.24	0.20	0.22	0.22	0.21	0.22
Color-based AM + color	0.34	0.36	0.37	0.37	0.37	**0.41**	0.40	0.38	0.38	0.38
Color-based AM	0.39	0.43	0.43	0.46	0.44	0.48	0.47	0.46	**0.49**	0.45
Successor based	0.34	0.32	0.33	0.33	0.34	0.34	**0.35**	0.33	0.33	0.34
DFS colors	0.21	0.21	0.22	0.19	0.21	0.22	0.22	0.22	**0.23**	0.21
	Support Vector Machine									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color	0.39	0.49	0.49	0.53	0.54	0.56	0.55	0.61	**0.62**	0.53
Color-based AM + color	0.33	0.42	0.46	0.50	0.52	0.55	0.54	**0.56**	0.54	0.49
Color-based AM	0.19	0.20	0.20	0.20	**0.21**	0.20	0.20	**0.21**	0.18	0.20
Successor based	0.40	0.47	0.53	0.56	0.57	0.59	0.62	**0.66**	0.65	0.56
DFS colors	0.38	0.45	0.50	0.51	0.53	0.57	0.58	0.56	**0.61**	0.52
	Multilayer perceptron									
	10%	20%	30%	40%	50%	60%	70%	80%	90%	Mean
Node-based AM + color	0.43	0.51	0.52	0.55	0.58	0.53	0.54	0.60	**0.62**	0.54
Color-based AM + color	0.40	0.44	0.45	0.50	0.52	0.54	**0.55**	0.54	0.54	0.50
Color-based AM	0.43	0.47	0.52	0.53	0.56	0.58	**0.60**	0.57	0.57	0.54
Successor based	0.45	0.49	0.52	0.56	0.59	0.56	**0.62**	**0.62**	0.60	0.56
DFS colors	0.41	0.48	0.49	0.54	0.51	0.55	**0.58**	0.56	**0.58**	0.52

Overall, the greatest mean classification accuracy achieved the proposed method in all experiments was 74%. This was done in experiment 2, when combined with SB and RF. Using the notation proposed in section 4.4, the exact implementation was:

$$\{MGM, \{OF\}, SB, RF\}$$

This implementation also achieved the highest single classification accuracy of 79%, when 90% of the dataset was used for training and 10% was used as test.

5.4.1 Classification accuracy boxplots

For each combination of feature vector generation and classifier sub-methods, nine different models are created and evaluated, and the mean accuracy is calculated. Boxplots may be used to show the differences in the distributions of results between different combinations. A boxplot, sometimes referred to as a whisker plot, is a standardized method for plotting distributions of data, and to compare different distributions. This method for plotting data was published by Tukey [100] in 1977. Boxplots highlight several features of distributions, including central tendency, dispersion, clustering, skewness and extremes [101].

In the boxplots, each distribution is summarized by six values: the minimum and maximum value, the interquartile range, the mean value and the median value. The minimum and maximum values are represented by the lower and upper whiskers, respectively. The interquartile range is represented by the box. The lower end of box displays the value that divides the first and second quartile of data, while the upper end divides the third and forth quartile. As a result, the box contains the middle two quartiles, or the middle 50%, of the data in the distribution. Finally, the mean value is marked by an "X", and the median value by a horizontal line.

Figures 24 and 25 show the boxplots of the classification results from experiment 1 and 2, respectively. For both plots, the first four boxes represent implementations that used Random Forrest (RF), the next four Naive Bayes (NB), the next four Support Vector Machine (SVM), and the final four Multilayer perceptron (MLP). The orange boxes represent implementations using Color-based adjacency matrix with color (CAMc), yellow boxes represent Color-based adjacency matrix without color (CAM), light blue boxes represent Successor based (SB), and green boxes represent Depth first search colors (DFSc). Figure 24 show that in experiment 1, RF with SB or DFSc had the best performance with regards to classification accuracy, and that SVM with CAM had the worst. We can see the same results for experiment 2 in figure 25.

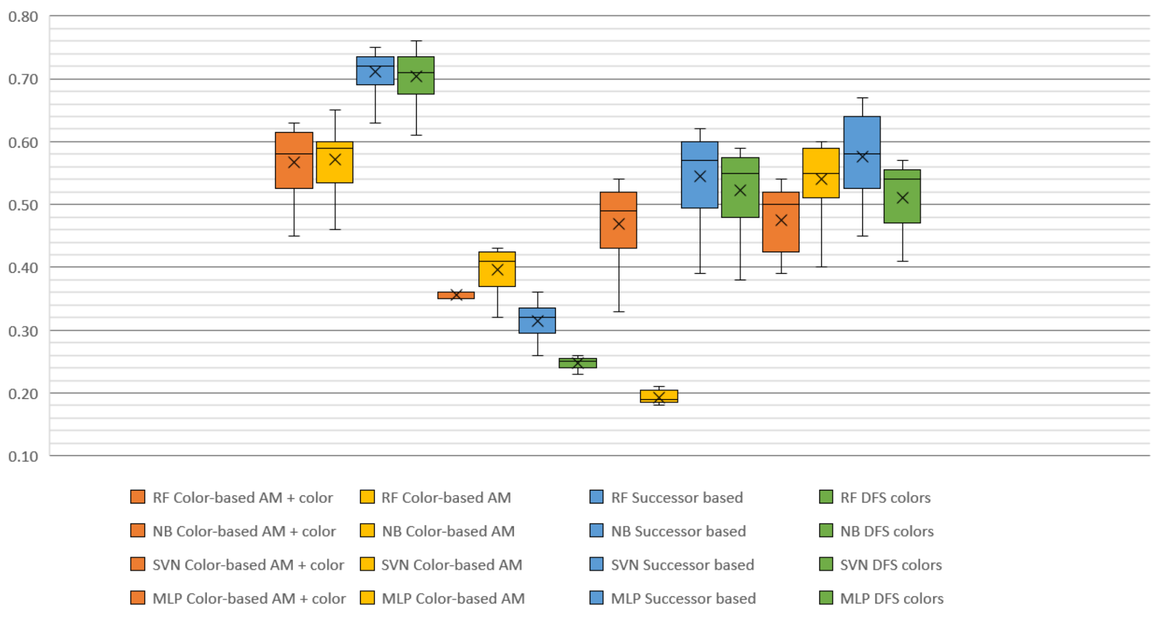


Figure 24: Boxplot of the classification results from experiment 1

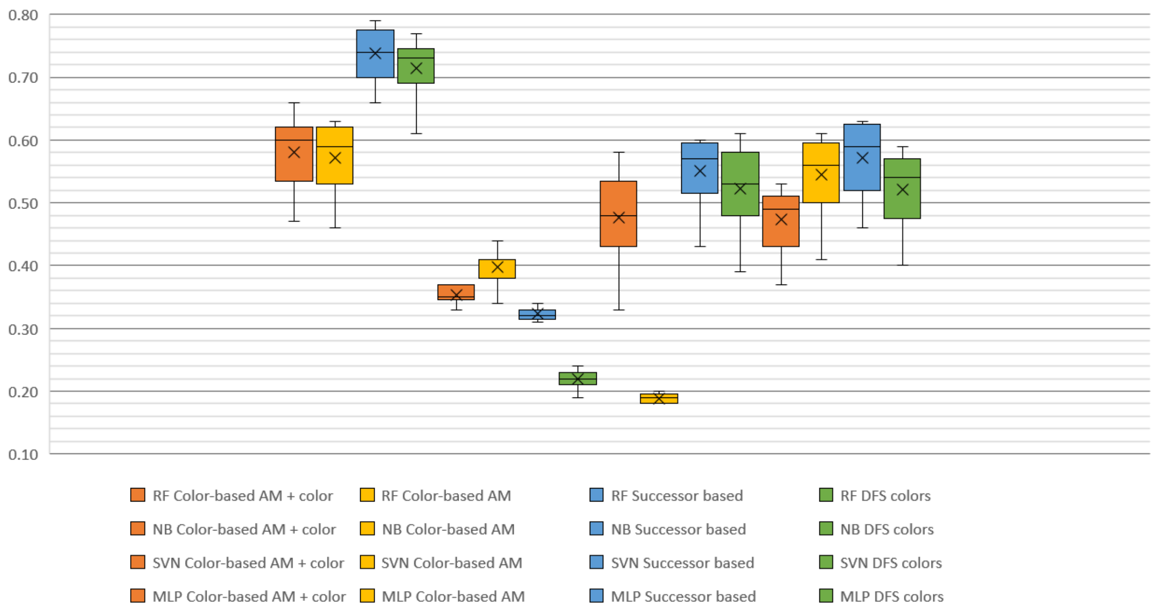


Figure 25: Boxplot of the classification results from experiment 2

Figures 26 and 27 show the boxplots of the classification results from experiment 3 and 4, respectively. For both plots, the first five boxes represent implementations that used RF, the next five NB, the next five SVM, and the final five MLP. The dark blue boxes represent implementations using NAMc, orange boxes represent CAMc, yellow boxes represent CAM, light blue boxes represent SB, and green boxes represent DFSc. Both figures show that RF, when combined with NAMc, SB or DFSc performed best with regards to classification accuracy, and NB with NAMc or DFSc and SVM with CAM performed worst.

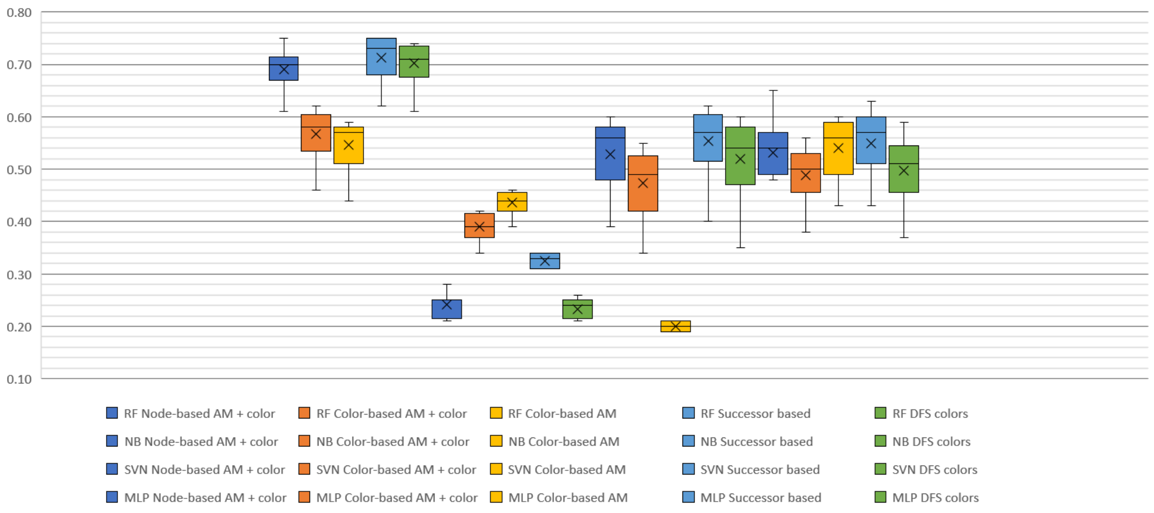


Figure 26: Boxplot of the classification results from experiment 3

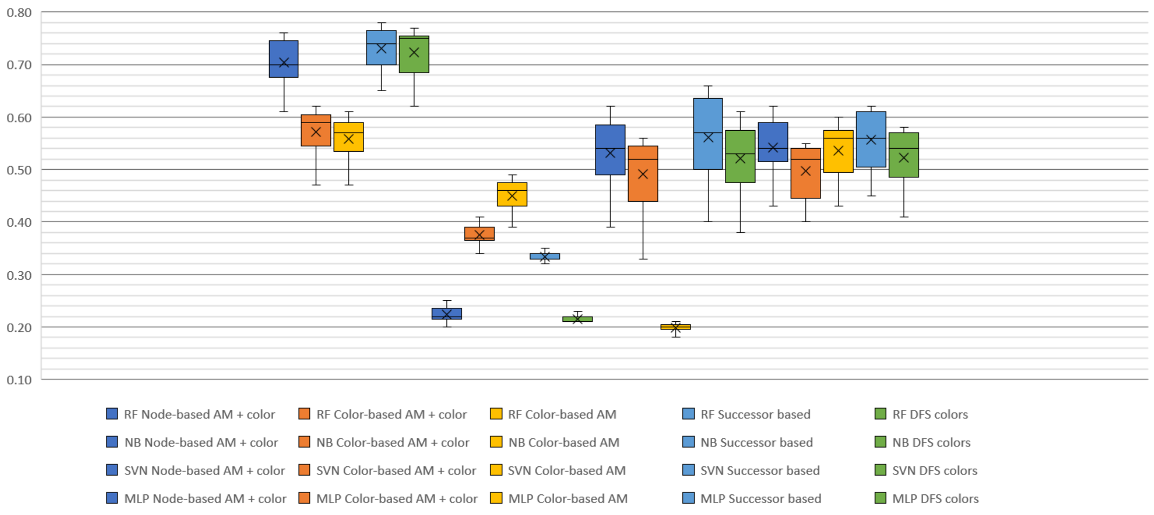


Figure 27: Boxplot of the classification results from experiment 4

Finally, we see similar distributions between the experiments for the same part 3 and 4 sub-methods. For example, when comparing the four implementations that used RF and DFSc, we see that the distributions are very similar. This is the case for all evaluated combinations of part 3 and 4 sub-methods. Additionally, implementations based on RF performed on average best, while methods based on NB performed on average worst. The other two classifiers, SVM and MLP, performed somewhere in between, with the exception of implementations combining SVM and CAM, which performed worst in all experiments.

6 Discussion

This chapter presents a discussion of the theoretical and practical implications of the results. In this thesis, we set out to determine if and how control flow graphs recovered using static analysis could be used for multinominal performance. Chapter 3 presents a theoretical method to answer this, and describes a framework of sub-modules that implements the method. Chapter 4 presents the implementation of the proposed method, evaluating a sub-set of the sub-methods presented in the previous chapter, and chapter 5 presents the results of the experiments.

First, we discuss and compare the main results of the thesis, followed by a discussion of the work conducted in this thesis with regards to the research questions presented in section 1.5. Following this are the theoretical and practical implications of the results, before finally, we present recommendations for future research.

For simplicity, the different sub-methods are referred to by the abbreviations given in section 4.4.

6.1 Main results

This thesis proposes a method for multinomial classification of malware, using static analysis to recover behavioural information, and using this information for machine learning based classification. The experiments conducted in this thesis evaluated a sub-set of possible implementations of the method as a proof-of-concept, using a dataset divided into malware families. As the dataset consisted of ten classes, a classifier relying on random guessing would achieve a classification accuracy of 10%. In comparison, all implementations of our method achieved higher accuracy than this, which indicate that they are all, to a certain degree, capable of classifying malware families. For the rest of this section, we will discuss and compare the evaluated sub-methods in each part.

For part 1, only one sub-method, MGM, was evaluated in the experiments, and comparing different control flow graph recovery sub-methods is left for future work.

Four different sub-methods were evaluated in part 2: experiment 1 used no pro-processing, experiment 2 used OF, experiment 3 used LN and experiment 4 used both OF and LN. The goal of this part was to determine if the control flow graphs could in some way be adjusted, to improve the performance of the method. Experiment 1 served as a baseline for comparison. The first sub-method, OF, orders function in the order in which they are called in the original sample, under the assumption that this will better represent the behaviour of the sample. This sub-method was included in experiment 2 and 4. The results shows only minor improvements of classification accuracy when OF was applied. Using highest mean classification accuracy of each experiment, we see an improvement from 71% in experiment 1, to 74% in experiment 2, and from 71% in experiment 3, to 73% in experiment 4. In almost all combinations, including OF improved classification accuracy, but only slightly.

The second sub-method, LN, limits the number of nodes in the graphs to a predetermined number. In the experiments, the mean number of nodes in all samples (after removing non-representative graphs in part 1) was used as the limit: 433. This number was chosen as it is dependent on the dataset used, and not chosen arbitrarily. The goal of this sub-method is to reduce the amount of information, and preferably included the set of nodes that are best suited for classification. In our experiments, we used the first 433 nodes from each graph. Without also including OF, the nodes selected are determined by the order in which the functions are recovered in part 1. Experiment 3 therefore shows how our method performs when the number of nodes is reduced, without considering which nodes should be included. In experiment 4, OF was applied before LN, such that the set of nodes included in each sample corresponds to the first nodes in each sample. The results show only minor improvements between experiments 3 and 4, implying that this sub-method is only able to capture nodes that are slightly better suited for classification, and that there may exist a set of nodes that is even better suited. Analyzing which nodes this may be and developing an automatic method for filtering them is left for future work.

Comparing the results of experiment 1 and 3 shows how reducing the number of nodes, by applying LN, had little effect on the results. All combinations performed almost exactly the same with regards to classification accuracy, with and without LN. However, it is important to note that information was reduced in both cases, as the feature vectors generated without LN were to long for classification. As stated in 4.4.7, all feature vectors in experiment 1 and 2 were truncated to the mean length, while all feature vectors in experiments 3 and 4 were padded to the maximum length. For SB and DFSc, this difference had no effect on the length of the resulting vectors, except for minor differences caused by rounding down when applying LN, which is why the feature vectors are slightly shorter in experiment 1 and 2. For CAMc and CAM, this significantly affects the content of the feature vectors, but had little effect on the classification accuracies in our experiments. When applying LN, we reduced the number of different colors, and therefore also reduced the size of the adjacency matrix. This is shown in section 5.3, where the calculated mean number of colors decreases from 53.89 and 53.86 in experiments 1 and 2, to 44.94 and 43.40 in experiments 3 and 4. However, in experiments 1 and 2, we made a similar reduction in information, by truncating the feature vectors. As a result, combinations including CAMc and CAM performed similarly with and without LN, which indicates that the same amount of information was included. For NAMc, applying LN enables us to included it in the experiments, as without, the resulting feature vectors exceeded hardware limitations. Additionally, as NAMc generates vectors whose length is dependent on the number of nodes, combining it with LN results in all feature vectors being of the same length.

Overall, including LN lets us generate feature vectors using fewer nodes, which may improve computational efficiency. However, for CAMc and CAM, longer feature vectors must be used for classification, which reduces efficiency. This is dependent on the exact implementation, and has been left for future work.

For part 3 of the proposed method, five different sub-methods were included. CAMc, CAM, SB and DFSc were evaluated in all experiments, while NAMc was only evaluated in experiments 3 and

4. All these sub-methods relied on colors to abstract the content of nodes, and we will therefore start the discussion of the part 3 sub-methods with this topic. Colors are used to convert the assembly instructions within each node into a fixed length representation. This also adds resistance to some obfuscation techniques, such as register reassignment and code substitution. The drawback of colors is that they reduce the amount of information available for the rest of the method. Our results does, however, show that the proposed method is capable of classifying malware, which implies that the colors contain sufficient information.

NAMc was based on the original method in [7]. This sub-method uses colors to represent nodes, and an adjacency matrix to encode structural information form the graphs. It generates feature vectors whose lengths are only dependent on the number of nodes, resulting in equal length, aligned feature vectors when all graphs have the same number of nodes. The main drawback of this sub-method is that it generates very long and sparse feature vectors. CAMc and CAM were suggested as possible improvements to NAMc, to generate shorter, less sparse feature vectors, but with the drawback of having vector lengths depend on the number of different colors, making them unaligned. Our results showed that CAMc and CAM performed worse than NAMc in almost all cases, indicating that they were less capable of capturing behavioural information. Comparing CAMc to CAM also shows that the explicit inclusion of colors in the feature vectors had a positive effect on the classification accuracy, although minor.

SB and DFSc used a fundamentally different technique to represent structural information. Nodes are ordered in the order they are executed in the original sample, using the depth first search algorithm. DFSc simply orders the nodes, and adds the color of each node to the feature vector. SB aims to improve DFSc, by adding additional structural information about the successors of each node. The results show that in all cases, SB performed better than DFSc with regards to classification accuracy, although only slightly better when combined with RF. This improvement does, however, come at the cost of having five times longer feature vectors, which may negatively affect the computational efficiency of the classification method.

NAMc performed best among the adjacency matrix based methods, while SB performed best of the depth first search methods. Comparing their best mean results in both experiment 3 and 4, NAMc resulted in a classification accuracy of 69% against SB with 71% in experiment 3, and 70% against 73% in experiment 4. SB did however achieve these results with significantly shorter feature vectors, with lengths of 2165 for SB against 187922 of NAMc, making this sub-method more efficient. NAMc generates very sparse feature vectors, while SB packs the information more densely. As they resulted in very similar classification accuracies, this indicates that they are both capable of capturing and representing similar amounts of information from the graphs.

Finally, for part 4 of the proposed method, four different classifiers were evaluated. The results showed that RF performed significantly better than the other classifiers in all experiments, while NB performed worse. The main advantage of RF is its ability to generalize, by using the majority vote of several tree-based classifiers to determine the class. The numbers of nodes and edges presented in figures 20, 21, 22 and 23, as well as the color distributions in figure 16, indicates a low intra-class similarity between the original samples in the dataset. The former shows that there are great differences in the number of nodes and edges in sample of the same class, and the latter shows that each class contains a great number of different colors. This lack of similarity may be caused by, for instance, obfuscation or different compiler versions [68]. This increases the necessity of generalization, and may be the reason why RF performed better than the other classifier sub-methods.

NB was the worst performing classifier in all our experiments. We did, however, see from our classifier validation method that implementations using NB performed almost equally well with both small and large percentages of training data, which indicates that adding additional training data does not improve the modeled probability distribution, and that the features may be too dependent on each other for NB. This conclusion is supported by the fact that different code structures depend on each other to implement the features of a program, and therefore should not be seen as independent.

SVM and MLP performed somewhere in the middle of the two other classifiers, with the notable exception of implementations using CAM and SVM, which performed significantly worse in all experiments. CAM is the only part 3 sub-method that does not directly include the colors in the feature vectors, which may be why this combination did so poorly. Adding colors in CAMc, we see a significant improvement in classification accuracy, which further strengthens this hypothesis.

It is also worth noting that no machine learning based pre-processing or feature selection techniques were applied to the feature vectors in this thesis. Both [45] and [102] included these techniques in their experiments, which improved classification accuracy. Applying these techniques may have similar effects on our results, and is left for future research.

6.2 Research questions

The research conducted in this thesis aims to develop a method for multinomial malware classification based on static analysis that includes behavioural information. In particular, a method based on control flow graphs that contains this information. This method consists of several parts, and was therefore designed to be highly modular for easier development and comparison. Several sub-methods were suggested for each part, some of which were compared in the experiments.

The first research question stated in section 1.5 asks how control flow graphs that include behavioural information can be recovered from malware using static analysis. This question was addressed by part 1 of the method. An important aspect of this is that the resulting control flow graph must correctly represented the behavior of the analyzed sample. Initial studies showed that this topic had already been addressed extensively in other research, and several of the techniques are implemented in different software. This indicates that state-of-the-art software is capable of recovering representative control flow graphs from software. These techniques does however sometimes fail when applied to malware, as malware often use obfuscation techniques to hinder or prevent analysis. As a result, a minimum requirement for the resulting graphs was applied, as graphs from obfuscated samples does not correctly represent the behavior of the malware.

No previous research was found to discuss the different representations of the control flow graphs. We present two different representations, single graph mode and multiple graphs mode, and discusses their advantages and drawbacks. During development, several problems were encountered with single graph mode, and due to time limitations, only multiple graphs mode were included in the experiments.

The second research question addressed how pre-processing can be applied to control flow graphs to improve the performance of the method. Two different pre-processing techniques and the combination of them where evaluated, as well as a baseline without pre-processing. The classification results showed that both techniques had minimal effects on the classification accuracy of the method. This is however interesting, as it shows that reducing the amount number of nodes (by applying LN in experiments 3 and 4) available when generating the feature vectors does not significantly affect classification accuracy. However, as was described in section 4.2.5, the length of the feature vectors must at some point be limited, for practical reasons. Our experiments therefore showed the point at which information is limited does not significantly affect classification accuracy. It does however affect the amount of information required to create the feature vectors and their resulting lengths, which affects the computational resources required at each part. Experiment 1 and 2 used more information to generate the feature vectors, but shorter lengths were used for classification, while experiment 3 and 4 used the opposite. This affects the total computational resources required by the method, but this was not investigated in detail in this thesis.

The third question asks how control flow graphs can be converted into feature vectors that includes behavioural information. Initially, we researched different methods for graph based classification. These methods compare graphs directly without converting them to some other representation, but suffer from high computational resource requirements. We therefore changed out focus to methods that converts the graphs into feature vectors, such that traditional machine learning based classifiers could be used. A requirement was that the conversion must include both information within the nodes and structural information (represented by the edges). Five different sub-methods, based on two different techniques, was evaluated and compared. The sub-methods differ in how they represent the structural information, but all used colors to represent the information within the nodes. The colors offer an abstract representation of instructions within each node, and are more resistant to several common obfuscation techniques. Analysis of the color distributions showed significant differences between the classes, supporting the claim that colors can be used to represent instructions.

Structural information is represented either using an Adjacency Matrix, or in the ordering. The original method, NAMc, is base on the one developed by Kruegel et al. in [7], using an Adjacency Matrix to represent structural information, and was included for comparison. CAMc and CAM was developed in an attempt to improve the original method, by using a less sparse matrix. Our results showed that, while they produced shorter feature vectors, they resulted in significantly lower classification accuracy. CAMc produced slightly better results than CAM, indicating that the inclusion of colors only had a minimal effect on the results in most cases. On the other hand, the two methods that included structural information in node ordering, specifically using the depth first search algorithm, produced similar results as the original method, while using scientifically shorter feature vectors. This makes both generating the feature vectors and classification significantly more computationally efficient. SB resulted in slightly better accuracy than DFSc, implying that the additional structural information was useful for the classifier.

The final research question asks what performance can be achieved with regards to classification accuracy, given a set of evaluated implementations of the proposed method. While not exhaustive, we were able to evaluate several different sub-methods and combinations, giving a reasonable indication of the classification accuracy of the proposed method. Out of the four included classification sub-methods, the one based on RF performed best in all experiments, with the highest mean classification accuracy of 74% when combined with SB in experiment 2. The other classifiers achieved at most: 45% for NB in experiment 4 with CAM, 56% for SVM in experiment 4 with SB, and 58% for MLP in experiment 1 with SB. An important observation that was made with regards to the classification accuraices is that the choice of classification sub-method had the greatest affect on the results. This indicates that all of the evaluated feature vector generation methods was able generate representative feature vectors, and that the pre-processing sub-methods only made minor improvements to the graphs.

The overall low accuracy of the implementations based on NB in part 4 indicates that the features are not conditionally independent. This is supported by the fact that the nodes and edges that are represented in the feature vectors are not independent, and depend on each other to represent different code-structures in the executable binary. As a result, we do not consider NB to be a good fit for the method proposed in this thesis.

6.3 Theoretical considerations

The method proposed in this thesis attempted to classify malware by recovering behavioral and structural information using static analysis techniques. Typically, malware classification methods based on static analysis exclude behavioral information, and focus on string matching of content or metadata. Behavioral information is often the focus of classifiers based on dynamic analysis, which are computationally less efficient and require a lot of setup to get the malware to execute in controlled environments. Incident responders can therefore benefit from the proposed method, as it considers behavioral information without the drawbacks of dynamic analysis.

Converting the graphs into feature vectors enable the use of traditional machine leaning methods. Graph learning methods may have some advantages, with regards to learning structural information. On the other hand, these methods are less researched, and often do not have functioning implementations. By converting graphs into feature vectors that include representations of the structural information, we combine the advantages of both learning methods.

The proposed method is designed as a framework that can be used to create and evaluate different implementations of the method. During development, standardized interfaces between the parts were developed. These interfaces enable the use of different sub-methods within each part, such that a sub-method in one part may be replaced with another of the same part, without making changes to the other parts (with some limitations, as described in section 3.5. New techniques can be developed as sub-methods, and evaluated with other sub-methods and compared to other implementations. This enables other researcher to easily develop and compare new techniques.

6.4 Practical implications

One of the main considerations when evaluating a new malware classification method is the dataset used. The dataset must be representative of the real world, and be correctly labelled. Additionally, using an open, available dataset enables others to re-evaluate the proposed method, and compare their methods to ours. The dataset used in this thesis is a sub-set of the one generated by the Testimon Research Group. This dataset was created from open sources, and has been used in other research.

The class-set used in this thesis was malware families, based on the definition given in section 3.7. Malware families was chosen as the class-set in this thesis over malware types, which are also included as labels in the dataset. This decision was made because information related to malware families is more prevalent when using static analysis than information relevant to malware type. Our method uses static analysis to recover the code structure, which is highly dependent on how the malware achieves its goal, and less so on what exactly that goal is. The method may be able to classify malware types, as there will be some similarities between samples of the same type, but will results in a lower classification accuracy.

A major problem that when encountered when implementing the experiments was recovering representative control flow graphs. Several samples resulted in graphs containing very few nodes and edges. This indicates that the graph had not been correctly recovered, that the resulting graph did not represent the sample, and that it could not be used in classification. To counter this, a minimum node requirement was set, such that any graph with fewer nodes would be excluded from the rest of the experiment. The threshold was set to 20 nodes, so as to not exclude too many samples in the dataset, based on figure 17. No further analysis was conducted in this thesis to determine if this is the optimal threshold, and how well graphs of length 20 represents their original samples.

The primary reason for samples resulting in small, unrepresentative graphs was obfuscation. IDA Pro recovers graphs by performing automatic, static analysis, and is unable to determine, for instance, the targets of indirect jumps. The analysis process therefore stopes when a jump of this kind is encountered, and outputs the graph it has recovered up until that point. Graph recovery may be improved, but there will always be some indirect jumps whose target that cannon be determined without executing the malware sample, such as those where the target is based on information obtained from a source outside the sample. Obfuscation is therefore, as with most methods based on static analysis, the main problem of this method when analyzing new samples.

6.5 Further research

The methods for multinomial malware classification proposed in this thesis builds on the original method by Kruegel et al. [7], in an attempt to improve it and evaluate it using a modern dataset. Recovering control flow graphs using static analysis of malware, and using them to perform classification is not new, but the framework created in this thesis and the suggested sub-methods extends and standardizes existing methods. However, as mentioned earlier, the framework was developed to be highly modular, and several sub-methods was suggested for each part. There remains therefore several combinations of sub-methods left to evaluate, as well as other details of the method that can be further researched to improve classification and our understanding. This section presents identified topics that may be studied to further the research conducted in this thesis.

The first topic for further research are the additional sub-methods suggested for part 1, 2, 3 and 4, and described in sections 3.1.4, 3.2.4, 3.3.7 and 3.4.6, respectively. These sub-methods were not included in the experiments due to the time constraints of the thesis. Additionally, two of the sub-methods were attempted, single graph mode for part 1 and removing library functions for part 2, but were excluded due to problems encountered during implementation. Both should be evaluated, but the latter requires additional study to determine effective ways to detect library functions and further investigate how this affects the graphs.

As described in section 3.1.4, control flow graphs may also be recovered using dynamic analysis. This may be implemented as a part 1 sub-method, and combined with the other sub-methods evaluated in this thesis. Comparing the results of this implementation with the once evaluated in this thesis will determine how well our method, based on static analysis, performs against the same method based on dynamic analysis. The result of this comparison will improve our knowledge of the differences between automatic static and dynamic analysis.

A minimum node requirement was set to exclude graphs that were not representative of the sample from which they were recovered. This requirement can be further studied to determine how well it captures it objective. Additionally, some other requirement may be used, to determine if a graph is representative or not. The number of nodes or edges required may for instance be determined by the size of the original sample. Improving the requirement for analysis will help exclude unrepresentative graphs and include representative graph, and thereby improve classification accuracy.

The feature vectors generated in part 3 of the proposed method include both structural information and information about the nodes. Structural information is represented either by a adjacency matrix or in the ordering, and node information is represented by colors. Which one of these contributed most to the classification accuracy is a topic for further research. This was not specifically studied in this theses, but by comparing the results of the color-based adjacency matrix sub-methods with and without colors, we see that the explicit inclusion of colors did improve accuracy slightly.

Furthermore, the concept of colors as a abstract representation of the information within the nodes requires additional research. The instruction groups used in this thesis that make up the colors are based on the once used by Kruegel et al. in [7]. As described in section 4.1, there are several factors that affect how the colors are generated. Firstly, is the groups and number of groups used, secondly, the order of the groups, and finally, the which instructions are placed in which group. Determining optimal values for these parameters will improve classification, as the colors will better represent the information in the nodes.

Finally, a comparison of different class-sets should be performed, similarly to the one conducted in [43]. Determining if the method is capable of classifying malware types, and how well it does so compared to malware families will provide additional insight into the differences between the class-sets. As the proposed method is class-set agnostic, performing this experiment simply requires that class labels are reassigned based malware types. Additionally, this may be combined with the previous suggestion of using dynamic analysis to recover control flow graphs, to determine if those graphs better capture the behavioral information that malware types rely on.

7 Conclusion

The primary goal of this thesis was to develop a method for multinomial malware classification based on static analysis that included behavioural information. The method builds upon and extends an existing method by Kruegel et al. [7], whom used this method to detect and classify malicious worms. Control flow graphs are used to obtain both behavioural information, represented by the edges and structure of the graphs, and information within the nodes. In the original method, the authors recovered the control flow graphs of malware, converted the graphs into feature vectors and used them to perform classification. Our method extends theirs by suggesting and evaluating different pre-processing and feature vector generation methods, as well as using different machine learning based classifiers to perform multinomial classification.

A preliminary experiment was conducted to evaluate the original method, using a more modern dataset and our implementation to recreate the method created by Kruegel et al. In the experiment, we performed binomial classification, using a dataset of 1000 benign and 1000 malware samples, and achieved a mean classification accuracy of 89%. These results showed that the method was still relevant and could be used with a more general malware dataset, and should be further studied.

A framework was created to simplify development and evaluation of the method. This framework is designed to be highly modular, and standardizes malware classification methods based on control flow graphs by dividing the them into four parts. The parts are: control flow graph recovery, graph based pre-processing, feature vector generation, and classification. Different techniques have been suggested as sub-methods for different parts, and each sub-method may be exchanged by any other sub-method of the same part, with some limitations. A given implementation of the proposed method therefore consist of one sub-method from each part. The modularity of the framework enables easy development, evaluation and comparison of new techniques.

Several sub-methods have been suggested in this thesis, and a sub-set of the possible combinations of these have been evaluated through experiments. The experiments evaluated the combinations of one control flow graph recovery sub-method, four graph based pre-processing sub-methods, five feature vector generation sub-methods, and four classification sub-methods. In the end, 72 different combinations of sub-methods were evaluated, as eight of the methods were excluded due to technical limitations discussed in the thesis. The dataset used in the experiments was a sub-set of the one created by the Testimon Research Group. It consisted of 9823 malware samples divided into 10 classes. Each sample was labelled using malware families, and a discussion on different class-sets is provided in the thesis.

The experimental results present the classification accuracies of the experiments, as well as the intermediate results of each sub-method of the evaluated combinations. The results showed a great variation in classification accuracy from the different combinations of sub-methods. The highest mean classification accuracy achieved, by the implementations of our method evaluated in this thesis, was 74% in a 10-class classification problem.

In conclusion, this thesis has developed a method for multinomial malware classification based on static analysis that include behavioural information. Several different implementations of the method has been suggested, and a sub-set of these have been evaluated through experiments, using a large, modern dataset of malware. The experiments serves as a proof-of-concept for the method, and shows that it capable of classifying malware with high classification accuracy. As a result, this thesis has made a contribution to malware classification based on static analysis, and more specifically, those methods that aim to include behavioural information.

Bibliography

[1] Levin, B. & Simpson, D. Apr 2019. Malware names. https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/malware-naming. [Online; accessed 2019-05-16].

[2] Grunzweig, J. Apr 2013. Basic packers: Easy as pie. https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/basic-packers-easy-as-pie/. [Online; accessed 2019-05-17].

[3] Committee, T. et al. 1995. Tool interface standard (tis) executable and linking format (elf) specification version 1.2. *TIS Committee*.

[4] You, I. & Yim, K. Nov 2010. Malware obfuscation techniques: A brief survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, 297–300. doi:10.1109/BWCCA.2010.85.

[5] Bengoetxea, E., Larrañaga, P., Bloch, I., Perchant, A., & Boeres, C. 2002. Inexact graph matching by means of estimation of distribution algorithms. *Pattern Recognition*, 35(12), 2867 – 2880. Pattern Recognition in Information Systems. URL: http://www.sciencedirect.com/science/article/pii/S0031320301002321, doi:https://doi.org/10.1016/S0031-3203(01)00232-1.

[6] Software, S. Pe files. http://www.silurian.com/inspect/peformat.htm. [Online; accessed 2019-05-17].

[7] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., & Vigna, G. 2006. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, 207–226, Berlin, Heidelberg. Springer-Verlag. URL: http://dx.doi.org/10.1007/11663812_11, doi:10.1007/11663812_11.

[8] Cloutier, F. Feb 2019. x86 and amd64 instruction reference. https://www.felixcloutier.com/x86/. [Online; accessed 2019-05-18].

[9] Unknown. elf(5) - linux man page. https://linux.die.net/man/5/elf. [Online; accessed 2019-05-17].

[10] BBC. Jan 2016. Hackers caused power cut in western ukraine - us. https://www.bbc.com/news/technology-35297464. [Online; accessed 2019-05-16].

[11] Sikorski, M. & Honig, A. 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition.

[12] Mohurle, S. & Patil, M. 2017. A brief study of wannacry threat: Ransomware attack 2017. *International Journal of Advanced Research in Computer Science*, 8(5).

[13] Ladakis, E., Koromilas, L., Vasiliadis, G., Polychronakis, M., & Ioannidis, S. 2013. You can type, but you can't hide: A stealthy gpu-based keylogger. In *Proceedings of the 6th European Workshop on System Security (EuroSec)*.

[14] Organization, C. A. R. Caro. https://docs.microsoft.com/en-us/windows/security/threat-protection/intelligence/malware-naming. [Online; accessed 2019-05-16].

[15] Bishop, M. 2003. *Computer Security: Art and Science*. Addison-Wesley. URL: https://books.google.no/books?id=pfdBiJNfWdMC.

[16] Konstantinou, E. & Wolthusen, S. 2008. Metamorphic virus: Analysis and detection. *Royal Holloway University of London*, 15, 15.

[17] Pirscoveanu, R. S., Hansen, S. S., Larsen, T. M. T., Stevanovic, M., Pedersen, J. M., & Czech, A. June 2015. Analysis of malware behavior: Type classification using machine learning. In *2015 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, 1–7. doi:10.1109/CyberSA.2015.7166115.

[18] Zolkipli, M. F. & Jantan, A. Sep. 2010. Malware behavior analysis: Learning and understanding current malware threats. In *2010 Second International Conference on Network Applications, Protocols and Services*, 218–221. doi:10.1109/NETAPPS.2010.46.

[19] Mirkovic, J. & Reiher, P. April 2004. A taxonomy of ddos attack and ddos defense mechanisms. *SIGCOMM Comput. Commun. Rev.*, 34(2), 39–53. URL: http://doi.acm.org/10.1145/997150.997156, doi:10.1145/997150.997156.

[20] BBC. Jan 2019. Briton who knocked liberia offline with cyber attack jailed. https://www.bbc.com/news/uk-46840461. [Online; accessed 2019-05-16].

[21] Bokhari, S. N. Aug 1995. The linux operating system. *Computer*, 28(8), 74–79. doi:10.1109/2.402081.

[22] Chen, H., Mao, Y., Wang, X., Zhou, D., Zeldovich, N., & Kaashoek, M. F. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, 5:1–5:5, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/2103799.2103805, doi:10.1145/2103799.2103805.

[23] Bunten, A. 2004. Unix and linux based rootkits techniques and countermeasures. In *16th Annual First Conference on Computer Security Incident Handling, Budapest*.

[24] Farmer, D. & Venema, W. 2009. *Forensic Discovery*. Addison-Wesley Professional, 1st edition.

[25] Joy, J., John, A., & Joy, J. 2011. Rootkit detection mechanism: A survey. In *Advances in Parallel Distributed Computing*, Nagamalai, D., Renault, E., & Dhanuskodi, M., eds, 366–374, Berlin, Heidelberg. Springer Berlin Heidelberg.

[26] Ismail, A., Hajjar, M., & Hajjar, H. 2008. Remote administration tools: A comparative study. *Journal of Theoretical & Applied Information Technology*, 4(2).

[27] TeamViewer. 2019. Teamviewer. https://www.teamviewer.com/. [Online; accessed 2019-05-16].

[28] Kennedy, J. & Satran, M. May 2018. Remote desktop protocol. https://docs.microsoft.com/en-us/windows/desktop/termserv/remote-desktop-protocol. [Online; accessed 2019-05-17].

[29] Chien, E. 2005. Techniques of adware and spyware. In *the Proceedings of the Fifteenth Virus Bulletin Conference, Dublin Ireland*, volume 47.

[30] Gazet, A. Feb 2010. Comparative analysis of various ransomware virii. *Journal in Computer Virology*, 6(1), 77–90. URL: https://doi.org/10.1007/s11416-008-0092-2, doi:10.1007/s11416-008-0092-2.

[31] Paquet-Clouston, M., Haslhofer, B., & Dupont, B. 2018. Ransomware payments in the bitcoin ecosystem. *CoRR*, abs/1804.04080. URL: http://arxiv.org/abs/1804.04080, arXiv:1804.04080.

[32] Huang, D. Y., Aliapoulios, M. M., Li, V. G., Invernizzi, L., Bursztein, E., McRoberts, K., Levin, J., Levchenko, K., Snoeren, A. C., & McCoy, D. May 2018. Tracking ransomware end-to-end. In *2018 IEEE Symposium on Security and Privacy (SP)*, 618–631. doi:10.1109/SP.2018.00047.

[33] CYBERSEC. Aug 2018. Remember fancy bear? https://www.secjuice.com/fancy-bear-review/. [Online; accessed 2019-05-17].

[34] Sathyanarayan, V. S., Kohli, P., & Bruhadeshwar, B. 2008. Signature generation and detection of malware families. In *Information Security and Privacy*, Mu, Y., Susilo, W., & Seberry, J., eds, 336–349, Berlin, Heidelberg. Springer Berlin Heidelberg.

[35] Bennett, J. T., Moran, N., & Villeneuve, N. 2013. Poison ivy: Assessing damage and extracting intelligence. *FireEye Threat Research Blog*.

[36] Fredrikson, M., Jha, S., Christodorescu, M., Sailer, R., & Yan, X. May 2010. Synthesizing near-optimal malware specifications from suspicious behaviors. In *2010 IEEE Symposium on Security and Privacy*, 45–60. doi:10.1109/SP.2010.11.

[37] Mohaisen, A. & Alrawi, O. 2013. Unveiling zeus: automated classification of malware samples. In *Proceedings of the 22nd International Conference on World Wide Web*, 829–832. ACM.

[38] Binsalleeh, H., Ormerod, T., Boukhtouta, A., Sinha, P., Youssef, A., Debbabi, M., & Wang, L. Aug 2010. On the analysis of the zeus botnet crimeware toolkit. In *2010 Eighth International Conference on Privacy, Security and Trust*, 31–38. doi:10.1109/PST.2010.5593240.

[39] Kennedy, J., Robertson, C., Satran, M., & LeBLanc, M. Mar 2019. Pe format. https://docs.microsoft.com/en-us/windows/desktop/debug/pe-format. [Online; accessed 2019-05-17].

[40] Molkenthin, B. Jan 2002. Pe file format offsets. http://www.sunshine2k.de/reversing/tuts/tut_pe.htm. [Online; accessed 2019-05-17].

[41] Østbye, M. O. 2017. Multinomial malware classification based on call graphs.

[42] Thompson, K. June 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6), 419–422. URL: http://doi.acm.org/10.1145/363347.363387, doi:10.1145/363347.363387.

[43] Banin, S. & Dyrkolbotn, G. O. 2018. Multinomial malware classification via low-level features. *Digital Investigation*, 26, S107 – S117. URL: http://www.sciencedirect.com/science/article/pii/S1742287618301956, doi:https://doi.org/10.1016/j.diin.2018.04.019.

[44] Islam, A. C., Yamaguchi, F., Dauber, E., Harang, R. E., Rieck, K., Greenstadt, R., & Narayanan, A. 2015. When coding style survives compilation: De-anonymizing programmers from executable binaries. *CoRR*, abs/1512.08546. URL: http://arxiv.org/abs/1512.08546, arXiv:1512.08546.

[45] Chitrakar, A. S. 2013. Author identification from text-based communications: Identifying generalized features and computational methods.

[46] Egele, M., Scholte, T., Kirda, E., & Kruegel, C. March 2008. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2), 6:1–6:42. URL: http://doi.acm.org/10.1145/2089125.2089126, doi:10.1145/2089125.2089126.

[47] Collberg, C., Thomborson, C., & Low, D. 1997. A taxonomy of obfuscating transformations.

[48] Eilam, E. 2011. *Reversing: secrets of reverse engineering*. John Wiley & Sons.

[49] Cifuentes, C. & Emmerik, M. V. May 1999. Recovery of jump table case statements from binary code. 192–199. doi:10.1109/WPC.1999.777758.

[50] Xu Chen, Andersen, J., Mao, Z. M., Bailey, M., & Nazario, J. June 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 177–186. doi:10.1109/DSN.2008.4630086.

[51] Branco, R. R., Barbosa, G. N., & Neto, P. D. 2012. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat*.

[52] Kennedy, J. & Satran, M. May 2018. Windows api index. https://docs.microsoft.com/en-us/windows/desktop/apiindex/windows-api-list. [Online; accessed 2019-05-18].

[53] Chenke, L., Feng, Y., Qiyuan, G., Jiateng, Y., & Jian, X. July 2017. Anti-reverse-engineering tool of executable files on the windows platform. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 1, 797–800. doi:10.1109/CSE-EUC.2017.158.

[54] Intel. Sep 2016. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.

[55] Seshadri, A., Luk, M., Perrig, A., Doorn, L. v., & Khosla, P. 2006. Externally verifiable code execution. *Communications of the ACM*, 49(9), 45–49.

[56] Balzarotti, D., Cova, M., Karlberger, C., Kirda, E., Kruegel, C., & Vigna, G. 2010. Efficient detection of split personalities in malware. In *NDSS*.

[57] Allen, F. E. 1970. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, 1–19, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/800028.808479, doi:10.1145/800028.808479.

[58] Ryder, B. G. May 1979. Constructing the call graph of a program. *IEEE Trans. Softw. Eng.*, 5(3), 216–226. URL: http://dx.doi.org/10.1109/TSE.1979.234183, doi:10.1109/TSE.1979.234183.

[59] Oh, N., Shirvani, P. P., & McCluskey, E. J. March 2002. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1), 111–122. doi:10.1109/24.994926.

[60] Hu, X., Chiueh, T.-c., & Shin, K. G. 2009. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 611–620, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/1653662.1653736, doi:10.1145/1653662.1653736.

[61] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., & Vigna, G. May 2016. Sok: (state of) the art of war: Offensive techniques in binary analysis. 138–157. doi:10.1109/SP.2016.17.

[62] Hex-Rays. Ida pro. https://www.hex-rays.com/products/ida/index.shtml. [Online; accessed 2019-05-18].

[63] Xu, L., Sun, F., & Su, Z. 2010. Constructing precise control flow graphs from binaries.

[64] Kronjee, J., Hommersom, A., & Vranken, H. 2018. Discovering software vulnerabilities using data-flow analysis and machine learning. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES 2018, 6:1–6:10, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/3230833.3230856, doi:10.1145/3230833.3230856.

[65] Monga, M., Paleari, R., & Passerini, E. 2009. A hybrid analysis framework for detecting web application vulnerabilities. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, IWSESS '09, 25–32, Washington, DC, USA. IEEE Computer Society. URL: http://dx.doi.org/10.1109/IWSESS.2009.5068455, doi:10.1109/IWSESS.2009.5068455.

[66] Kinder, J. Static analysis of x86 executables. Technical report, Technische Universität Darmstadt, 2010.

[67] Cifuentes, C. & Van Emmerik, M. 02 1999. Recovery of jump table case statements from binary code. 192 – 199. doi:10.1109/WPC.1999.777758.

[68] Sand, L. A. 2012. Information-based dependency matching for behavioral malware analysis.

[69] Kruegel, C., Robertson, W., Valeur, F., & Vigna, G. 2004. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, 18–18.

[70] Kinable, J. 01 2010. Malware detection through call graphs.

[71] Lee, J. B., Rossi, R., & Kong, X. 2018. Graph classification using structural attention. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &#38; Data Mining*, KDD '18, 1666–1674, New York, NY, USA. ACM. URL: http://doi.acm.org/10.1145/3219819.3219980, doi:10.1145/3219819.3219980.

[72] Cook, D. J. & Holder, L. B. Subdue - graph based knowledge discovery. http://ailab.wsu.edu/subdue/. [Online; accessed 2018-12-09].

[73] Bray, T. 2014. The javascript object notation (json) data interchange format.

[74] Tarjan, R. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 146–160. URL: https://doi.org/10.1137/0201010, arXiv:https://doi.org/10.1137/0201010, doi:10.1137/0201010.

[75] Kil, C., Jun, J., Bookholt, C., Xu, J., & Ning, P. Dec 2006. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 339–348. doi:10.1109/ACSAC.2006.9.

[76] Wilson, R. J. 1979. *Introduction to graph theory*. Pearson Education India.

[77] Shannon, C. E. July 1948. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3), 379–423. doi:10.1002/j.1538-7305.1948.tb01338.x.

[78] Skiena, S. S. *Sorting and Searching*, 103–144. Springer London, London, 2008. URL: https://doi.org/10.1007/978-1-84800-070-4_4, doi:10.1007/978-1-84800-070-4_4.

[79] Pearl, J. 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[80] Kononenko, I. & Kukar, M. 2007. Chapter 3 - machine learning basics. In *Machine Learning and Data Mining*, Kononenko, I. & Kukar, M., eds, 59 – 105. Woodhead Publishing. URL: http://www.sciencedirect.com/science/article/pii/B9781904275213500034, doi:https://doi.org/10.1533/9780857099440.59.

[81] Breiman, L. 01 2001. Random forests, machine learning 45. *Journal of Clinical Microbiology*, 2, 199–228.

[82] Crisci, C., Ghattas, B., & Perera, G. 2012. A review of supervised machine learning algorithms and their applications to ecological data. *Ecological Modelling*, 240, 113 – 122. URL: http://www.sciencedirect.com/science/article/pii/S0304380012001081, doi:https://doi.org/10.1016/j.ecolmodel.2012.03.001.

[83] John, G. H. & Langley, P. 2013. Estimating continuous distributions in bayesian classifiers. *CoRR*, abs/1302.4964. URL: http://arxiv.org/abs/1302.4964, arXiv:1302.4964.

[84] Zhang, H. 01 2004. The optimality of naive bayes. volume 2.

[85] Cortes, C. & Vapnik, V. Sep 1995. Support-vector networks. *Machine Learning*, 20(3), 273–297. URL: https://doi.org/10.1007/BF00994018, doi:10.1007/BF00994018.

[86] Hinton, G. E. 1989. Connectionist learning procedures. *Artificial Intelligence*, 40(1), 185 – 234. URL: http://www.sciencedirect.com/science/article/pii/0004370289900490, doi:https://doi.org/10.1016/0004-3702(89)90049-0.

[87] Zwietering, P., Aarts, E., & Wessels, J. 1991. The design and complexity of exact multilayered perceptrons. *International Journal of Neural Systems*, 02(03), 185–199. URL: https://doi.org/10.1142/S0129065791000170, arXiv:https://doi.org/10.1142/S0129065791000170, doi:10.1142/S0129065791000170.

[88] VirusTotal. 2019. Virustotal. https://www.virustotal.com/. [Online; accessed 2019-05-18].

[89] The CCC, C. C. C. Mar 2010. Ccc dataset 2009. https://www.iwsec.org/mws/2009/en_about.html. [Online; accessed 2019-05-19].

[90] Fraze, D. 2016. Cyber grand challenge (cgc). URL: https://www.darpa.mil/program/cyber-grand-challenge.

[91] Wang, R. Oct 2015. Performance / cpu load. https://github.com/angr/angr/issues/36. [Online; accessed 2019-05-18].

[92] Lukan, D. Jan 2013. Ida: Cross references / xrefs. https://resources.infosecinstitute.com/ida-cross-references-xrefs/. [Online; accessed 2019-05-18].

[93] Team, T. U. Upx the ultimate packer for executables. https://upx.github.io/. [Online; accessed 2019-05-18].

[94] Hex-Rays. Ida f.l.i.r.t. technology: In-depth. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml. [Online; accessed 2019-05-18].

[95] Kernighan, B. W., Ritchie, D. M., Tondo, C. L., & Gimpel, S. E. 1988. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, NJ.

[96] Anderson, B., Quist, D., Neil, J., Storlie, C., & Lane, T. 11 2011. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, 7, 247–258. doi:10.1007/s11416-011-0152-x.

[97] Bragen, S. R. 2015. Malware detection through opcode sequence analysis using machine learning.

[98] The Python Software Foundation. 2019. The python programming language. https://www.python.org/. [Online; accessed 2019-05-18].

[99] Hex-Rays. 2019. Idapython documentation. https://www.hex-rays.com/products/ida/support/idapython_docs/. [Online; accessed 2019-05-18].

[100] Tukey, J. 1977. *Exploratory Data Analysis*. Addison-Wesley series in behavioral science. Addison-Wesley Publishing Company. URL: https://books.google.no/books?id=UT9dAAAAIAAJ.

[101] Banacos, P. 01 2011. Box and whisker plots for local climate datasets: Interpretation and creation using excel 2007/2010.

[102] Berg, P. E. 2011. Behavior-based classification of botnet malware.

[103] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.

A IDA Pro Graph recovery scripts

Appendix A presents the two Python 2.7 scripts used to recover the control flow graphs using IDA Pro. Listing A.1 contains the script *ida_multiple.py*, and listing A.2 contains the script *ida_recover_graphs.py*. The first script is a helper script used to read the dataset, execute the second script in parallel and handle errors. The second script is executed by IDA Pro in text mode, and uses the IDAPython API [99] to interact with IDA Pro. It waits until IDA has finished the analysis, before recovering control flow graphs of functions and program sections using the Depth First Search algorithm. IDA Pro refers to program sections as *segments*, which is why this name is used in the scripts. This script also includes the part 2 sub-method Ordered Functions, as this requires information from IDA to execute.

The scripts assumes:

- *ida_recover_graphs.py* is located at "C:\script\ida_recover_graphs.py"
- "C:\experiment\output\" exists, and can be used to store graphs in JSON [73] files
- "C:\experiment\error\" exists, and can be used to store error files

ida_recover_graphs.py can also be executed on a single malware sample, using the command:

C:\Program FilesC\IDA 7.2\idat64.exe -c -A -S"C:\script\ida_recover_graphs.py LABEL LABEL_ID ORDERED_FUNCTIONS" PATH_TO_SAMPLE

Where:

- LABEL is the class label of the sample
- LABEL_ID is the numerical representation of the class label
- ORDERED_FUNCTIONS is a flag (0 or 1) to indicate if functions should be ordered
- PATH_TO_SAMPLE is the path to the malware sample

Listing A.1: Python script ida_multiple.py

```
import os
import sys
import json
from glob import glob
from multiprocessing import Pool
import subprocess

# Reads the class labels by reading directory names in the dataset
# Arguments:
#   path: path to the root of the dataset
# Return value:
#   Mapping of {CLASS_LABEL: NUMERICAL_REPRESENTATION}
```

```
def read_lables(path):
    l = {}
    for i, p in enumerate(glob(path + "\\*")):
        l[p.split("\\")[-1]] = i
    return l

# Generator function that yields pairs of (BINARY_PATH,CLASS_LABEL)
# Arguments:
#   path: path to the root of the dataset
# Yields:
#   pairs of (BINARY_PATH, CLASS_LABEL)
def read_dataset(path):
    # Iterate over the class directories
    for p in glob(path + "\\*"):
        # Get the label of the current class directory
        l = p.split("\\")[-1]

        # Iterate over malware samples in class directory
        for b in glob(p + "\\*"):
            # Ignore unwanted files (IDA DB, for instance)
            if not "." in b:
                # Yield pairs of (BINARY_PATH, CLASS_LABEL)
                yield (b, l)

# Helper function used to remove any leftover files after analysis
# Arguments:
#   binary_path: path to the analyzed binary
def cleanup(binary_path):
    for fl in glob(binary_path + ".*"):
        os.remove(fl)

# Function used to execute IDA in a sparate process
# Arguments:
#   args: touple containg the arguments required to execute IDA and keep status:
#       number: the sequential number of the binary.
#       binary: path to the malware sample
#       binary_label: the class label of the malware sample
#       label_id: Numerical representation of class label
#       oredered_functions: flag to indicate if functions should be ordered
# Return value:
#   error: True if error, False if success
def execute_ida(args):
    # Unpack the arguments
    number, binary, binary_label, label_id, oredered_functions = args
    new_name = binary_label + "_" + binary.split("\\")[-1]

    # The command to be executed
    cmd = \
    'C:\\Program␣Files\\IDA␣7.2\\idat64.exe␣-c␣-A␣-S"C:\script\ida_recover_graphs.py␣'\
    + binary_label + "␣"\
    + str(label_id) + '␣'\
    + str(oredered_functions) + '"␣'\
    + binary

    try:
        # Execute command and wait for completion, cleanup, and log success
        subprocess.call(cmd)
        cleanup(binary)
        print("[F]␣File␣number:␣" + str(number) + ",␣File␣name:␣" + new_name)
        return False
    except Exception as e:
        # Log any exeptions that occured
        print("[E]␣File␣number:␣" + str(number) + ",␣File␣name:␣" + new_name\
            + ",␣Error:␣" + str(e))
        return True

# Main function of the script
# Arguments:
#   input_path: path to root of dataset
#   oredered_functions: flag to indicate that functions should e ordered or not
def main(input_path, oredered_functions):
    # "label_map.json" contains a mapping of:
    # {CLASS_LABEL: NUMERICAL_REPRESENTATION}
    # Use existing map if found, or create a new one
    if os.path.isfile("label_map.json"):
        with open("label_map.json", "r") as f:
            lables = json.load(f)
    else:
        lables = read_lables(input_path)
        with open("label_map.json", "w") as f:
            json.dump(lables, f)

    # Create a list of touples used when executing IDA.
    # Each touple represent a malware sample
    dataset = [
        (i, b, l, lables[l], oredered_functions)
        for i, (b, l) in enumerate(read_dataset(input_path))
    ]

    # Counters for statistics
    num_failed = 0
    num_comp = 0

    # Create a pool of 8 worker-processes used to execute IDA in parallel
    pool = Pool(processes=8, maxtasksperchild=1)

    # Map the dataset to the function, using the workers to execute 8
    # instances of IDA in parallel
    for err in pool.imap_unordered(execute_ida, dataset):
        # Logs completed samples. Counts number of successes and failed.
        # Outputs every failed and every tenth success
        if err:
            num_failed += 1
            print("[I]␣Completed:␣" + str(num_comp) + ",␣Number␣of␣failed:␣"\
                + str(num_failed))
        else:
            num_comp += 1
            if num_comp % 10 == 0:
                print("[I]␣Completed:␣" + str(num_comp)\
                    + ",␣Number␣of␣failed:␣" + str(num_failed))

    print("Finished!")
    print("[I]␣Completed:␣" + str(num_comp) + ",␣Number␣of␣failed:␣"\
        + str(num_failed))

# Get commandline args and call main function
if __name__=='__main__':
    input_path = sys.argv[1]
    oredered_functions = int(sys.argv[2])

    main(input_path, oredered_functions)
```

Listing A.2: Python script ida_recover_graphs.py

```
import idaapi
import idautils
import idc
import json
import sys

# Helper function to get the start address of a function from its name
def get_addr_from_name(name):
    for func_addr in idautils.Functions():
        if idaapi.get_func_name(func_addr) == name:
            return func_addr

# Helper function to get the information from each node,
# using IDA to decode the instructions
def get_info(node):
    instructions = []
    for inst_addr in idautils.Heads(node.start_ea, node.end_ea):
        inst = idautils.DecodeInstruction(inst_addr)
        if inst:
            instructions.append(inst.get_canon_mnem())

    return {
            "id": node.id,
            "start": node.start_ea,
            "end": node.end_ea,
            "type": node.type\
                if not (node.type == 0 and len(list(node.succs())) == 2)\
                    else 3,
                "instructions": instructions
            }

# Helper function to get a list of imported functions
def get_lib_funcs():
    l = []
    def callback(ea, name, ord):
        l.append(name)
        return True

    nimps = idaapi.get_import_module_qty()

    for i in xrange(0, nimps):
        idaapi.enum_import_names(i, callback)
    return l

# Function used to recover the control flow graph of a function
# Arguments:
#   func: function object
#   func_name: name of the function
#   imports: list of names of imported functions
# Return value:
#   dict-object containing nodes, edges and metadata of function
def get_cfg_of_func(func, func_name, imports):
    # Get the first node
    fc = idaapi.FlowChart(func)
    first = list(fc)[0]

    nodes = []
    node_ids = []
    edges = []

    # Depth first search to iterate over and store nodes and edges
    stack = [first]
    while stack:
        n = stack.pop()
        if n.id not in node_ids:
            node_ids.append(n.id)
            ni = get_info(n)
            nodes.append(ni)
            for s in n.succs():
                si = get_info(s)
                edges.append((ni, si))
                stack.append(s)

    # Convert nodes into mapping of {NUMBER: NODE_INFO}
    # node["id"] is the global unique id of each node,
    # while NUMBER is unique to function and sequential
    nodes_dict = {}
    inv_nodes_dict = {}
    for i, node in enumerate(nodes):
        nodes_dict[i] = node
        inv_nodes_dict[node["id"]] = i

    # Invers node map is used to replace global ids with NUMBER
    # to reduce size and simplify lookup for later
    edges_list = []
    for f, t in edges:
        edges_list.append((
            inv_nodes_dict[f["id"]],
            inv_nodes_dict[t["id"]]
        ))
    # Return dict-object containing nodes, edges and metadata of function
    return {
        "func_name": func_name,
        "nodes": nodes_dict,
        "edges": edges_list,
        "is_lib": bool(func.flags & idaapi.FUNC_LIB),
        "imported": func_name in imports
    }

# Function used to recover the control flow graph of a segment
# Arguments:
#   start_addr: start address of the segment
#   end_addr: end address of the segment
#   sclass: segment class, as determined by IDA
#   seg_name: name of the segment
# Return value:
#   dict-object containing nodes, edges and metadata of segment
def get_cfg_of_segment(start_addr, end_addr, sclass, seg_name):
    # Get the first node
    fc = idaapi.FlowChart(bounds=(start_addr, end_addr))
    try:
        first = list(fc)[0]
    except:
        # Return if no nodes or edges were found in segment
        return {
            "seg_name": seg_name,
            "nodes": {},
            "edges": [],
            "sclass": sclass
        }

    nodes = []
    node_ids = []
    edges = []

    # Depth first search to iterate over and store nodes and edges
    stack = [first]
    while stack:
        n = stack.pop()
        if n.id not in node_ids:
            node_ids.append(n.id)
            ni = get_info(n)
            nodes.append(ni)
            for s in n.succs():
                si = get_info(s)
                edges.append((ni, si))
                stack.append(s)

    # Convert nodes into mapping of {NUMBER: NODE_INFO}
    # node["id"] is the global unique id of each node,
    # while NUMBERs is unique to segment and sequential
    nodes_dict = {}
    inv_nodes_dict = {}
    for i, node in enumerate(nodes):
        nodes_dict[i] = node
        inv_nodes_dict[node["id"]] = i

    # Invers node map is used to replace global ids with NUMBERs
    # to reduce size and simplify lookup
    edges_list = []
    for f, t in edges:
        edges_list.append((
            inv_nodes_dict[f["id"]],
            inv_nodes_dict[t["id"]]
        ))
    # Return dict-object containing nodes, edges and metadata of segment
    return {
        "seg_name": seg_name,
        "nodes": nodes_dict,
        "edges": edges_list,
        "sclass": sclass
    }

# Wait untill IDA has finished the automatic analysis of the malware sample
idaapi.autoWait()

# Get the arguments of the script

# Class label of the malware sample
label = idc.ARGV[-3]

# Numeric representation of the class label
label_id = int(idc.ARGV[-2])

# Flag to indicate that functions should be ordered or not
oredered_functions = int(idc.ARGV[-1])
```

```
# Set output and error files
outfile = "C:\\experiment\\output\\" + label + "_"\
    + idaapi.get_root_filename() + ".json"
errfile = "C:\\experiment\\error\\" + label + "_"\
    + idaapi.get_root_filename() + ".json"

# Create output object
out = {"target": label_id}

# A try block around the script is used to simplify error-handeling
try:
    # Get a list of imported functions
    imports = get_lib_funcs()
    # Get a list of all functions
    funcs = [idaapi.get_func(fa) for fa in idautils.Functions()]

    # Used to hold recovered graphs
    out_functions = {}
    out_segments = {}

    # If the program contains any functions, use them to recover graphs
    if funcs:
        if oredered_functions:
            # The Depth First Search (DFS) Algorithm is used to order
            # the functions in the order they are executed. Every instruction
            # is considered a node, and the Xrefs functionality of IDA is used
            # to iterate through the instructions to find any references to
            # other functions. Functions are are stored in the order they are
            # encountered, using the start address of each function in the list.
            ordered_func_addrs = []

            # Get the entrypoint(s) of the executable to start DFS
            starts = [s[1] for s in idautils.Entries()]
            if not starts:
                raise AttributeError("STARTERROR:␣No␣start␣find␣in␣sample")

            for start_addr in starts:
                stack = []
                discovered = []

                # Get the first function from the start address
                start_func = idaapi.get_func(start_addr)
                if start_func is not None and\
                        start_func.startEA not in ordered_func_addrs:
                    ordered_func_addrs.append(start_func.startEA)

                # Perform DFS over all instructions, appending new functions
                # to the list. References to other functions are prioritized
                # over references to the same function, as those are executed
                # first.
            stack.append(start_addr)
            while stack:
                addr = stack.pop()
                if addr not in discovered:
                    discovered.append(addr)
                    xrefs = list(idautils.XrefsFrom(addr))
                    if len(xrefs) > 1:
                        cur_func = idaapi.get_func(addr)
                        if cur_func is not None:
                            cur_func_addr = idaapi.get_func(addr).startEA
                        else:
                            cur_func_addr = 0
                        o = []
                        for xref in reversed(xrefs):
                            func = idaapi.get_func(xref.to)
                            if func is not None:
                                func_addr = func.startEA
                                if func_addr != cur_func_addr and\
                                        func_addr not in ordered_func_addrs:
                                    ordered_func_addrs.append(func_addr)
                                o.insert(0, xref.to)
                            else:
                                o.append(xref.to)
                        stack.extend(o)
                    elif xrefs:
                        stack.append(xrefs[0].to)

        # Add funcs that were not discovered during DFS.
        # (Usually indirect or obfuscated calls)
        for func in funcs:
            if func.startEA not in ordered_func_addrs:
                ordered_func_addrs.append(func.startEA)

        # Get function objects from function addresses
        funcs = [idaapi.get_func(fa) for fa in ordered_func_addrs]

    # Recover the control flwo graph of each function
    for i, func in enumerate(funcs):
        # Get the name of the function
        f_name = idaapi.get_func_name(func.startEA)

        # Recover the control flow graph and append it to the output object
        g = get_cfg_of_func(func, f_name, imports)
        out_functions[i] = g
else:
    # If no functions are found, use segments to recover graphs

    seen_segs = []
    # Recover the control flwo graph of each function
    for i, seg_start in enumerate(idautils.Segments()):
            # Get segment object and name
            seg = idaapi.getseg(seg_start)
            seg_name = idc.SegName(seg_start)

            # Recover the control flow graph of the segment and append it
            # to the output object
            g = get_cfg_of_segment(seg.startEA, seg.endEA, seg.sclass, seg_name)
            out_segments[i] = g
            seen_segs.append(seg_start)

        # Additionally, recover control flow graphs from the entrypoints,
        # if they do not point to any of the segments.
        for (_, start_addr, _, name) in idautils.Entries():
            # Get segment object and name
            seg = idaapi.getseg(start_addr)
            seg_name = idc.SegName(seg_start)

            # Avoid duplicate segments
            if seg is None or seg.startEA in seen_segs:
                continue

            end_addr = seg.endEA

            # recover the control flow graph of the segment and append it
            # to the output object
            g = get_cfg_of_segment(start_addr, end_addr, seg.sclass, name)
            i += 1
            out_segments[i] = g

    # Append functions and segments to the output
    out["functions"] = out_functions
    out["segments"] = out_segments

    # Open file to write successfully recovered graphs to
    o = open(outfile, "w")
except Exception as e:
    # Append the error and open error file
    out["error"] = str(e)
    o = open(errfile, "w")

# Write output as JSON and close file
json.dump(out, o)
o.close()

# Terminate IDA
sys.exit()
```

B Feature vector generation script

Appendix B presents the Python 3 script used to implement the part 3 feature vector generation sub-methods, as well as the part 2 Limit Nodes sub-method and the minimum requirements for inclusion of samples. The script is presented in listing B.1. The script takes the JSON files produced by the scripts in appendix A as input, and outputs one CSV file for each specified feature vector generation sub-method. The script also outputs dataset statistics.

Listing B.1: Python script generate_feature_vectors.py

```
import sys
import json

from collections import OrderedDict
from glob import glob

# Function implementing the sub-method Node-based Adjacency Matrix with color.
def build_node_matrix_vector(nodes, color_nodes, edges, include_color):
    size = len(nodes)
    matrix = []
    for _ in range(size):
        matrix.append([0]*size)

    for f, t in edges:
        matrix[f][t] = 1

    if include_color:
        for fid in range(len(matrix)):
            matrix[fid].append(color_nodes[fid])

    vector = []
    for row in matrix:
        vector += row

    return vector

# Function implmenting the sub-methods Color-based Adjacency Matrix
# with and without color.
# The inclusion of colors are determined by the flag "include_color"
def build_color_matrix_vector(nodes, color_nodes, edges, include_color):
    unique_colors = set(color_nodes.values())
    size = len(unique_colors)
    matrix = {}

    if edges:
        for fid, tid in edges:
            fnc = color_nodes[fid]
            tnc = color_nodes[tid]

            if not fnc in matrix:
                matrix[fnc] = {tnc: 1}
            elif not tnc in matrix[fnc]:
                matrix[fnc][tnc] = 1
            else:
                matrix[fnc][tnc] += 1
    else:
        for nid in nodes.keys():
            ncl = color_nodes[nid]
            matrix[ncl] = {}


    empty_size = size
    if include_color:
        empty_size += 1

    final_matrix = []
    for fcol in unique_colors:
        if not fcol in matrix:
            final_matrix.append([0]*empty_size)
            continue

        row = []
        for tcol in unique_colors:
            if tcol in matrix[fcol]:
                row.append(matrix[fcol][tcol])
            else:
                row.append(0)

        if include_color:
            row.append(fcol)
        final_matrix.append(row)

    vector = []
    for row in final_matrix:
        vector += row

    return vector

# Function implementing the sub-method Depth First Search with color.
def build_dfs_vector(nodes, color_nodes, edges, include_color):
    vector = []
    for nid, node in nodes.items():
        vector.append(color_nodes[nid])
    return vector
```

```
# Function implementing the Successor Based sub-method.
def build_successor_vector(nodes, color_nodes, edges, include_color):
    successor_dict = OrderedDict()
    for fid, tid in edges:
        if not fid in successor_dict:
            successor_dict[fid] = [tid]
        else:
            successor_dict[fid].append(tid)

    out_nodes = []
    for nid, node in nodes.items():
        if not nid in successor_dict:
            out_nodes.append([color_nodes[nid], 0, 0, 0, 0])
            continue
        successors = successor_dict[nid]

        tid = successors[0]
        suc1 = [tid, color_nodes[tid]]

        if len(successors) > 1:
            tid = successors[1]
            suc2 = [tid, color_nodes[tid]]
        else:
            suc2 = [0, 0]

        out_nodes.append([color_nodes[nid]] + suc1 + suc2)

    vector = []
    for row in out_nodes:
        vector += row

    return vector

# The 14 groups of instructions used to generate colors:
# 0     Data         Transfer mov instructions
# 1     String       x86 string operations
# 2     Arithmetic   incl. shift and rotate
# 3     Flags        access of x86 flag register
# 4     Logic        incl. bit/byte operations
# 5     LEA          load effective address
# 6     Test         test and compare
# 7     Float        floating point operations
# 8     Stack        push and pop
# 9     Syscall      interrupt and system call
# 10    Branch       conditional control flow
# 11    Jump         unconditional control flow
# 12    Call         function invocation
# 13    Halt         stop instruction execution

# List contains 14 lists, each one containing the instructions corresponding
# to one of the groups above.
# Omitted from this script due to length, and can be found in appendix C.
instruction_groups = []

# Function for calculating the color of every node, using the above groups
# Argument:
#   nodes: dict-object mapping {NODE_ID: NODE_OBJECT}
# Return value:
#   color_nodes: dict-object mapping {NODE_ID: NODE_COLOR}
def build_colors(nodes):
    color_nodes = OrderedDict()

    # Iterate over the nodes
    for nid, node in nodes.items():
        # Get the list of instructions in the node
        try:
            instructions = node["instruction_addrs"].values()
        except:
            instructions = node["instructions"]

        # Start with an empty color
        color = 0
        # Iterate over the instructions
        for inst in instructions:
            for i, group in enumerate(instruction_groups):
                if inst in group:
                    # Set the the bit in the color corresponding to the group
                    # that the instruction was found in
                    color = color | (1 << i)
                    break
            else:
                # Output instructions that has not been grouped yet
                print("UNKNOWN␣INSTRUCTION:", inst)

        # Append the color to the resulting mapping
        color_nodes[int(nid)] = color

    return color_nodes

# Wrapper function around the different feature vector generation functions.
# Also calculates the color of every node.
def to_vector(nodes, edges, structure_func, include_color):
    # Get the color mapping for the nodes
    color_nodes = build_colors(nodes)
    try:
        # Generate the feature vector using the spesified structure function
        vector = structure_func(nodes, color_nodes, edges, include_color)
    except Exception as e:
        # Debug info in case of errors
        print("Nodes:", nodes)
        print("Color␣nodes:", color_nodes)
        print("Edges:", edges)
        print("len(nodes):", len(nodes))
        print("len(edges):", len(edges))
        print("len(color_nodes):", len(color_nodes))
        raise

    return vector

# Helper function to calculate the mean number of nodes in the dataset,
# excluding samples with less than "threshold" number of nodes
def mean_num_nodes(files, threshold=20):
    node_numbers = []
    # Iterate over all files in the dataset
    for file in files:
        with open(file, "r") as f:
            data = json.load(f)
        num_nodes = 0
        # Calculate the number of nodes in the given sample
        for func in list(data["functions"].values())\
                + list(data["segments"].values()):
            if "nodes" in func:
                num_nodes += len(func["nodes"])

        if not num_nodes < threshold:
            node_numbers.append(num_nodes)

    return int(sum(node_numbers) / len(node_numbers))

# Main function of the script
# Arguments:
#   input_path: path to the directory containing the graphs
#   graph_mode: used to indicate the type of grahs (only "mgm" was used
#               in the experiments)
#   structures: which structural representation sub-methods to use when
#               generating feature vectors (multiple can be provided as csv).
#   filter: if equal to 1: will filter out library functions. Only used
#               during initial testing, and not in the experiments.
#   remove_packed: if equal to 1: samples identified as packed or containing
#               less than 20 nodes are removed
#   limit_nodes: if not 0: limit the number of nodes used when generating
#               the feature vectors to the number specified. If -1, use the
#               mean number of nodes in all samples. Padds with empty
#               nodes if too few in a sample.
def main(input_path, graph_mode, structures,\
            filter, remove_packed, limit_nodes):
    # Get paths to all JSON files containing graphs
    if input_path.endswith("/"):
        input_path += "*"
    else:
```

```
    input_path += "/*"
files = list(glob(input_path))

out = {
    "samples": [],
    "lables": [],
}

# The implemented feature vector generation sub-methods,
# including configuration required to run them
supported_methods = {
    "namc": (
                "NAMc",
                build_node_matrix_vector,
                open("vectors_10k_NAMc_" + str(filter)\
                    + "_" + str(remove_packed)\
                    + "_" + str(limit_nodes) + ".csv", 'w'),
                1,
                []
            ),
    "camc": (
                "CAMc",
                build_color_matrix_vector,
                open("vectors_10k_CAMc_" + str(filter)\
                    + "_" + str(remove_packed)\
                    + "_" + str(limit_nodes) + ".csv", 'w'),
                1,
                []
            ),
    "cam":  (
                "CAM",
                build_color_matrix_vector,
                open("vectors_10k_CAM_" + str(filter)\
                    + "_" + str(remove_packed)\
                    + "_" + str(limit_nodes) + ".csv", 'w'),
                0,
                []
            ),
    "sb":   (
                "SB",
                build_successor_vector,
                open("vectors_10k_SB_" + str(filter)\
                    + "_" + str(remove_packed)\
                    + "_" + str(limit_nodes) + ".csv", 'w'),
                1,
                []
            ),
    "dfsc": (
                "DFSc",
                build_dfs_vector,
                    open("vectors_10k_DFSc_" + str(filter)\
                        + "_" + str(remove_packed)\
                        + "_" + str(limit_nodes) + ".csv", 'w'),
                    1,
                    []
                ),
    }

    # Select sub-methods based on input (csv)
    methods = []
    for structure in structures.split(","):
        methods.append(supported_methods[structure])

    # If limit_nodes is -1, calculate the mean number of nodes of all samples
    if limit_nodes == -1:
        limit_nodes = mean_num_nodes(files)
        print("limit_nodes", limit_nodes, flush=True)

    # For statistics
    filtered_noNodes = 0
    filtered_lib = 0
    filtered_sclass = 0
    total_included = 0

    upx_removed = 0
    small_removed = 0
    total_removed = 0

    numNodes = []
    numEdges = []

    allNodesStats = {}

    # Checks if graph_mode is "mgm" (Multiple Graphs Mode). Other graph_modes
    # can be implemented using "elif graph_mode == NAME:" below this codeblock
    if graph_mode == "mgm":
        for file in files:
            # Read JSON file to get graphs
            with open(file, "r") as f:
                data = json.load(f)

            all_nodes = OrderedDict()
            all_edges = []

            offset = 0

            # Iterate over all graphs of functions and segments
            for func in list(data["functions"].values())\
                    + list(data["segments"].values()):
                # If filter is enabled, remove functions and segments
                # identified as from libraries
                if filter:
                    if not func["nodes"]:
                        filtered_noNodes += 1
                        continue

                    if "imported" in func and\
                            "is_lib" in func and\
                            "func_name" in func and\
                            func["func_name"] != "start" and\
                            (func["imported"] or func["is_lib"]):
                        filtered_lib += 1
                        continue

                    if "sclass" in func and not func["sclass"] in [1,2,4]:
                        filtered_sclass += 1
                        continue

                # Get a list of every node from every function and segment,
                # and assign them new, per-sample unique IDs
                nodes = func["nodes"]
                for nid, node in nodes.items():
                    nid = int(nid)
                    all_nodes[nid + offset] = node

                # Remap the edges to use the new IDs
                for from_id, to_id in func["edges"]:
                    all_edges.append(
                        (
                            from_id + offset,
                            to_id + offset
                        )
                    )

                offset += len(nodes)
                total_included += 1

            all_nodes_length = len(all_nodes)

            # Append number of nodes to dataset statistics
            if all_nodes_length in allNodesStats:
                allNodesStats[all_nodes_length] += 1
            else:
                allNodesStats[all_nodes_length] = 1

            # Remove sample if identified as packed
            to_remove = 0
            if remove_packed:
                for seg in data["segments"].values():
                    if "upx" in seg["seg_name"].lower():
```

```
            upx_removed += 1
            to_remove = 1

if not all_nodes_length:
    print("No␣nodes␣left␣after␣filter␣in:", file)
    to_remove = 1

# Remove sample if it contains less than 20 nodes
if remove_packed and all_nodes_length < 20:
    small_removed += 1
    to_remove = 1

# Needed for statistics, as samples may be removed for both reasons
if to_remove:
    total_removed += 1
    continue

# Limmit the number of nodes used in the feature vectors
# Edges to or from removed nodes are also removed
# Too few nodes: add empty nodes without edges untill enough
if limit_nodes:
    tmp = OrderedDict()
    for nid, node in all_nodes.items():
        if nid == limit_nodes:
            break
        tmp[nid] = node
    else:
        nid += 1
        for i in range(nid, limit_nodes):
            tmp[i] = {"instructions": []}
    all_nodes = tmp
    all_nodes_length = len(all_nodes)

    tmp = []
    for fid, tid in all_edges:
        if fid < limit_nodes and tid < limit_nodes:
            tmp.append((fid, tid))
    all_edges = tmp

# Append number of nodes and edges to dataset statistics
numNodes.append(all_nodes_length)
numEdges.append(len(all_edges))

# Generate feature vectors using the specified sub-methods
for name, structure_func, of, include_color, vf_lengths in methods:
    # Generate a feature vector using a sub-method
    vector = to_vector(all_nodes, all_edges,\
                        structure_func, include_color)

    # Append its length for statistics
```

```
                vf_lengths.append(len(vector))

                # Write class label and feature vector to csv file
                print(
                    ','.join(
                        [
                            str(v) for v in
                            [data["target"]] + vector
                        ]
                    ),
                    file=of,
                    flush=True
                )

    # Close output files
    for _, _, of, _, _ in methods:
        of.close()

    # Write dataset statistics to file
    with open("all_nodes_stats.txt", "w") as stats_file:
        for l, v in allNodesStats.items():
            print(l, v, file=stats_file)

    # Print feature vector generation statistics
    print("Done!")
    print("total_included:", total_included)
    print("filtered_noNodes:", filtered_noNodes)
    print("filtered_lib:", filtered_lib)
    print("filtered_sclass:", filtered_sclass)

    print()
    print("Removed␣samples:")
    print("UPX:", upx_removed)
    print("Too␣small:", small_removed)
    print("Total␣removed:", total_removed)

    print()
    print("Number␣of␣nodes␣and␣edges:")
    print("Min:", min(numNodes), min(numEdges))
    print("Max:", max(numNodes), max(numEdges))
    print("Avrage:", sum(numNodes)/len(numNodes), sum(numEdges)/len(numEdges))

    for name, _, _, _, vf_lengths in methods:
        print()
        print("Feature␣vector␣lengths␣for␣" + name + ":")
        print("Min:", min(vf_lengths))
        print("Max:", max(vf_lengths))
        print("Avrage:", sum(vf_lengths)/len(vf_lengths))

# Get commandline arguments and call main function
```

```
if __name__=='__main__':
    input_path = sys.argv[1]
    graph_mode = sys.argv[2]
    structures = sys.argv[3]
    filter = int(sys.argv[4])
    remove_packed = int(sys.argv[5])
    limit_nodes = int(sys.argv[6])

    main(input_path, graph_mode, structures,\
            filter, remove_packed, limit_nodes)
```

C Instruction groups used to generate colors

This appendix contains the instruction groups used when generating the colors for the nodes. The groups are presented in listing C.1, as a Python list, so that they can easily used with the script in appendix B or other scripts. The groups are described in table 3 in section 2.12.2.

Listing C.1: Instruction groups

```
instruction_groups = [
["mov", "xchg", "cmpxchg", "movz", "movzx", "movs", "movsx", "in", "out", "
    xlatb", "prefetchnta", "movhps", "movd", "fcmovnu", "fcmovnb", "fcmovbe"
    , "cmovb", "cmovbe", "cmove", "cmovg", "cmovl", "cmovle", "cmovno", "
    cmovnp", "cmovns", "cmovp", "cmovs", "cmova", "cmovge", "cmovne", "cmovo
    ", "movaps", "xlat", "cmovnb", "cmovnz", "cmovz", "fcmove", "fstp9", "
    movapd", "movdqa", "movhlps", "movlpd", "movlps", "movmskps", "movnti",
    "movntps", "movntq", "movq", "movss", "movups", "pmovmskb", "fcmovb", "
    fcmovnbe", "fcmovne", "fcmovu", "prefetcht0", "prefetcht1", "prefetchwt1
    ", "pextrw", "pinsrw", "vmovhps", "vmovlps", "vmovsldup", "cmpxchg8b", "
    movshdup", "vpinsrw", "rdmsr"],
["movsb", "lodsb", "stosb", "cmpsb", "scasb", "movsw", "lodsw", "stosw", "
    cmpsw", "scasw", "movsd", "lodsd", "stosd", "cmpsd", "scasd", "rep␣movsb
    ", "rep␣lodsb", "rep␣stosb", "rep␣cmpsb", "rep␣scasb", "rep␣movsw", "rep
    ␣lodsw", "rep␣stosw", "rep␣cmpsw", "rep␣scasw", "rep␣movsd", "rep␣lodsd"
    , "rep␣stosd", "rep␣cmpsd", "rep␣scasd", "repne␣movsb", "repne␣lodsb", "
    repne␣stosb", "repne␣cmpsb", "repne␣scasb", "repne␣movsw", "repne␣lodsw"
    , "repne␣stosw", "repne␣cmpsw", "repne␣scasw", "repne␣movsd", "repne␣
    lodsd", "repne␣stosd", "repne␣cmpsd", "repne␣scasd", "repe␣movsb", "repe
    ␣lodsb", "repe␣stosb", "repe␣cmpsb", "repe␣scasb", "repe␣movsw", "repe␣
    lodsw", "repe␣stosw", "repe␣cmpsw", "repe␣scasw", "repe␣movsd", "repe␣
    lodsd", "repe␣stosd", "repe␣cmpsd", "repe␣scasd", "outs", "ins", "lods",
     "stos", "scas"],
["add", "sub", "mul", "imul", "div", "idiv", "neg", "adc", "sbb", "inc", "
    dec", "shr", "shl", "sar", "sal", "shld", "shrd", "ror", "rol", "rcr", "
    rcl", "aaa", "aad", "aam", "aas", "xadd", "pmulhw", "pmulhuw", "das", "
    daa", "paddb", "psrad", "pmaxub", "psrlq", "packssdw", "packsswb", "
    packuswb", "pmaddwd", "pmaxsw", "pminsw", "pminub", "pmullw", "pmuludq",
     "punpckhbw", "punpckhdq", "punpckhwd", "punpcklbw", "punpckldq", "
    psadbw", "pshufd", "pshufw", "pslld", "psllq", "psllw", "psraw", "psrld"
    , "psrlw", "psubb", "psubd", "psubq", "psubsb", "psubsw", "psubusw", "
    psubw", "paddd", "paddq", "paddsb", "paddsw", "paddusb", "paddusw", "
    paddw", "pavgb", "pavgw", "sha1nexte", "vpaddusw", "vpshufhw", "vpsrld",
     "vpsubsw", "vpsubusw", "vpunpckhbw", "vpunpckldq", "vpunpcklwd", "
    vpaddw"],
["sti", "cli", "std", "cld", "stc", "clc", "cmc", "sahf", "lahf", "sete", "
    setz", "setg", "setge", "setl", "setne", "salc", "setb", "setalc", "
    setnl", "setnle", "setno", "setnz", "clts", "setbe", "setle", "setnb", "
    setnbe", "setnp", "setns", "seto", "setp", "sets"],
["and", "or", "xor", "not", "cbw", "cwde", "cwd", "cdq", "cqo", "lock␣adc",
     "lock␣and", "lock␣xor", "bsf", "bsr", "bswap", "xorpd", "xorps", "pxor"
    , "orpd", "orps", "pand", "pandn", "vorps"],
["lea", "les", "lds", "lfs", "lgs"],
["test", "cmp", "bt", "btc", "btr", "bts", "cmps", "cmpeqsd", "cmpltpd", "
    cmpltsd", "cmpps", "vpcmpeqw", "pcmpeqd", "pcmpeqb", "pcmpeqw", "pcmpgtb
    "],
["f2xm1", "fabs", "fadd", "faddp", "fbld", "fbstp", "fchs", "fclex", "fcom"
    , "fcomp", "fcompp", "fcomp5","fdecstp", "fdisi", "fdiv", "fdivp", "
    fdivr", "fdivrp", "feni", "ffree", "fiadd", "ficom", "ficomp", "fidiv",
    "fidivr", "fild", "fimul", "fincstp", "finit", "fist", "fistp", "fisub",
     "fisubr", "fld", "fld1", "fldcw", "fldenv", "fldenvw", "fldl2e", "
    fldl2t", "fldlg2", "fldln2", "fldpi", "fldz", "fmul", "fmulp", "fnclex",
     "fndisi", "fneni", "fninit", "fnop", "fnsave", "fnsavew", "fnstcw", "
    fnstenv", "fnstenvw", "fnstsw", "fpatan", "fprem", "fptan", "frndint", "
    frstor", "frstorw", "fsave", "fsavew", "fscale", "fsqrt", "fst", "fstcw"
    , "fstenv", "fstenvw", "fstp", "fstsw", "fsub", "fsubp", "fsubr", "
    fsubrp", "ftst", "fwait", "fxam", "fxch", "fxtract", "fyl2x", "fyl2xp1",
     "fcos", "fsin", "fprem", "fprem1", "fucom", "fucomip", "fisttp", "addps
    ", "rsqrtps", "fucomp", "cvtdq2ps", "cvtpi2ps", "cvtps2pd", "cvtps2pi",
    "cvtsd2si", "cvtsi2sd", "cvttps2pi", "cvttsd2si", "emms", "unpckhpd", "
    unpckhps", "unpcklpd", "unpcklps", "rcpps", "addpd", "addsd", "andnpd",
    "andnps", "andpd", "andps", "divps", "divsd", "mulpd", "mulps", "mulsd",
     "maxps", "maxss", "minps", "fucomi", "fucompp", "fcomi", "fcomip", "
    fstp1", "fstp8", "fxch7", "fsincos", "subpd", "subps", "subsd", "comisd"
    , "ucomisd", "ffreep", "shufpd", "shufps", "sqrtps", "sqrtsd", "ucomiss"
    , "vaddsd", "vaddsubpd", "vandnpd", "vsubps", "vcvtsi2sd", "vdivsd", "
    vinsertps", "vminss", "vmulps", "vunpckhps", "cvtdq2pd", "vunpcklpd"],
["push", "pop", "pushf", "popf", "pusha", "popa", "pushad", "popad", "
    pushal", "popal", "pushfd", "popfd", "pushaw", "pushfw", "fxch4", "popaw
    ", "popfw"],
["int", "int3", "rdtsc", "cpuid", "sidt", "sldt", "rsldt", "bound", "verr",
     "verw", "lar", "into", "lock", "icebp", "arpl", "rsm", "xabort", "
    xbegin", "bndldx", "bndstx", "getsec", "lldt", "lsl", "str", "femms", "
    fsetpm", "invd", "ldmxcsr", "sfence", "sgdt", "skinit", "smsw", "stmxcsr
    ", "svldt", "svts", "wrmsr", "wbinvd", "vmread", "vmwrite", "vpcext", "
    prefetch", "prefetchw", "rdpmc"],
["je", "jne", "jg", "jge", "ja", "jae", "jl", "jle", "jb", "jbe", "jo", "
    jno", "jz", "jnz", "js", "jns", "loop", "loope", "loopne", "jecxz", "jp"
    , "jnp", "jcxz", "jnb", "loopw", "loopwe", "loopwne"],
["jmp", "nop", "ljmp", "ud0", "ud1", "ud2"],
["call", "enter", "syscall", "sysenter", "enterw"],
["halt", "ret", "hlt", "wait", "leave", "iret", "iretd", "retn", "retf", "
    leavew", "sysexit", "sysret", "retfw", "retnw", "iretw", "pause"],
]
```

D Classification script

Appendix D contains in listing D.1 the Python 3 script implementing the part 4 classifier sub-methods. The script takes the CSV files containing the feature vectors generated by the script in appendix B, and outputs the classification accuracies of the selected classifiers using the selected validation method. The classification methods are implemented using Scikit-learn [103].

Listing D.1: Python script perform_classification.py

```
import sys
import csv

from sklearn.ensemble import RandomForestClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC

from sklearn.model_selection import cross_val_score, train_test_split

# Helper function that padds or truncates the feature vectors to
# the spesified length
def pad(X, length):
    new_X = []
    for i, x in enumerate(X):
        try:
            if len(x) > length:
                new_X.append(x[0:length])
            else:
                new_X.append(x + [0]*(length - len(x)))
        except:
            print("i:", i)
            sys.exit(0)

    return new_X

# Main function of the script
# Arguments:
#   input_path: path to the csv file containing the feature vectors
#   classifier: which classifier sub-methods to use (multiple can
#               be provided as csv).
#   length: the length of the feature vectors.
#           if 0: max length, others padded with zeroes.
#           if -1: mean length rounded down to neares integer,
#                  trunctated or padded with zeroes.
#           else: use the spesified length.
#   validation_method: Which validation method to use:
#                       5-fold crossvalidation or 10-percent splits.
def main(input_path, classifier, length, validation_method):
    # Holds the feature vectors befor adjusting lengths
    _X = []
    # Holds the class labels
    y = []

    # Reads the feature vectors
    with open(input_path, "r", newline='') as f:
        reader = csv.reader(f)
        for row in reader:
            y.append(row[0])
            _X.append([int(x) for x in row[1:]])

    print("Finished␣reading:␣" + input_path, flush=True)

    # Calculates the feature vector length if necessary
    if length == 0:
        length = max([len(x) for x in _X])
    elif length == -1:
        length = int(sum([len(x) for x in _X])/len(_X))

    # Adjust the feature vectors to the correct lengths
    X = pad(_X, length)

    # The implemented classifier sub-methods, with required
    # configurations and meta-parametres
    supported_classifiers = {
        "rf":  (
                    "Random␣Forrest",
                    RandomForestClassifier(
                        n_estimators=2000,
                        max_depth=50,
                        random_state=0
                    )
                ),
        "nb":  (
                    "Naive␣Bayes",
                    MultinomialNB()
                ),
        "svm": (
                    "Support␣Vector␣Machines",
                    SVC(
                        gamma='scale',
                        kernel="poly",
                        cache_size=10000,
                        random_state=0
                    )
                ),
        "mlp": (
                    "Multi-layer␣Perceptron",
                    MLPClassifier(
                        hidden_layer_sizes=(1000,1000,1000),
                        max_iter=10000,
                        tol=0.000001,
                        learning_rate_init=0.0001,
                        random_state=0
                    )
                ),
    }

    # Select sub-methods based on input (csv)
    classifiers = []
    for cl in classifier.split(","):
        classifiers.append(supported_classifiers[cl])

    # Perform classification using the selected classifiers and
    # validation method
    for name, clf in classifiers:
        if validation_method == "5fold":
            print()
            print("Running␣5-fold-validation␣of",
                    name, "classifier:", flush=True)
            scores = cross_val_score(clf, list(X), y, cv=5, n_jobs=4)

            print("Scores:", scores)
            print("Accuracy:␣%0.2f␣(+/-␣%0.2f)" %\
                    (scores.mean(), scores.std() * 2), flush=True)

        elif validation_method == "10%split":
            scores = []
            print()
            print("Running␣10%split-validation␣of",
                    name, "classifier:", flush=True)
            for i in range(10, 100, 10):
                X_train, X_test, y_train, y_test = train_test_split(
                    X, y, train_size=i/100, test_size=(100-i)/100
                )

                clf.fit(X_train, y_train)

                score = clf.score(X_test, y_test)
                scores.append(score)

                print("Accuracy␣for␣{}%␣training␣and␣{}%␣testing␣samples:␣{}"
                    .format(i, 100-i, "%0.2f" % score), flush=True)

            print()
            print("Avrage␣accuracy:␣%0.2f" %\
                        (sum(scores)/len(scores)), flush=True)
        print("Done", flush=True)

    # Get commandline args and call main function
    if __name__=='__main__':
        input_path = sys.argv[1]
        classifier = sys.argv[2]
        length = int(sys.argv[3])
        validation_method = sys.argv[4]

        main(input_path, classifier, length, validation_method)
```

Magnus Simonsen Håland

Multinomial malware classification using control flow graphs

NTNU
Norwegian University of
Science and Technology