Malware detection through opcode sequence analysis using machine learning

Simen Rune Bragen

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2015

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

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

Malware detection through opcode sequence analysis using machine learning

Simen Rune Bragen

2015/06/01

Abstract

To identify malware, most antivirus scanners use a combination of signature matching and heuristic-based detection. Signature matching compares only to previously known files, while heuristic-based detection looks for known system artifacts and previously known bad code patterns in a file. The obvious problem with this is that only known and partially known samples will be recognized. In this thesis we use reverse engineering to extract the assembly instructions from a given executable file. We chose to use only the opcodes, which are the part of the instruction that specifies the operation to be performed, in example *mov*.

By performing statistical analysis on the datasets, a significant difference between the opcodes in malware and benign files was found. Due to this, supervised and unsupervised machine learning approaches like artificial neural network, support vector machine, bayes net, random forest, k nearest neighbours, and self organizing map was used to look at the sequences of these instructions. The unknown files were classified as either malware or benign depending on the presence of, and number of occurrences of different sequences.

We show that by using only opcodes without operands (the rest of the instruction), malware can be distinguished from benign files. By using a sequence length of up to four opcodes, a classification accuracy of 95,58% was achieved. Our work contributes to the research field by proving that also obfuscated malware due to the use of packers is detected through this method. By using different classifiers and longer sequences than previous work, we also provide empirical evidence that the n-gram length has little influence on the performance. We used a sequence length of four, compared to previous work that focused on only one and two sequences.

Sammendrag

Antivirusprogrammer gjenkjenner vanligvis skadevare på to måter: Ved å sammenligne signaturen til en fil med andre kjente signaturer, eller ved å kjenne igjen skadelig kode i filen. Problemet med disse metodene er at bare kjente og delvis kjente filer blir oppdaget. I denne oppgaven bruker vi "reverse engineering" for å hente ut assembly-instruksjonene til en gitt fil. Vi ser på "opcodene", altså den delen av instruksjonen som sier hva slags oppgave som skal utføres. Et eksempel er *mov*.

Ved hjelp av analyse av dataene fant vi en signifikant forskjell mellom opcodene som blir brukt av skadevare og de som blir brukt av vanlige programmer. Videre brukte vi flere forskjellige maskinlæringsalgoritmer for å lære av sekvensene av disse opcodene. Nye filer ble klassifisert som enten skadevare eller vennligsinnet programmer.

Ved å bruke sekvenser opptil 4 i lengden viser vi at en nøyaktighet på 95,58 % kan oppnås. Vi viderefører tidligere arbeid innen samme fagfelt ved å vise at også obfuskert skadevare kan gjenkjennes på denne måten, og vi bruker lengre sekvenser enn det er gjort tidligere.

Preface

I would like to thank my supervisor, Professor Katrin Franke for providing excellent guidance and valuable input throughout the whole project. Further, I would like to thank my classmates Roar, Xiongwei and Lars for discussions and company during the hours at the lab. Thank you to everyone at the Testimon Forensics Group for valuable feedback during our monthly meetings as well. Finally, I would like to thank my family for always challenging me and encouraging me to keep learning more.

Contents

Abstract . . . . **i**
Sammendrag . . . . **ii**
Preface . . . . **iii**
Contents . . . . **iv**
List of Figures . . . . **vii**
List of Tables . . . . **viii**
1 Introduction **1**
1.1 Keywords 1
1.2 Topic covered by the project 1
1.3 Problem description 1
1.4 Justification, motivation and benefits 1
1.5 Research questions 2
1.6 Scope and Contributions 2
1.7 Thesis outline 3
2 Malware identification **4**
2.1 Different types of malware 4
2.1.1 Virus 5
2.1.2 Worm 5
2.1.3 Bot 6
2.1.4 Rootkit 6
2.1.5 Backdoor 6
2.1.6 Trojan horse 6
2.2 Obfuscation techniques 7
2.2.1 Encryption 7
2.2.2 Packers 8
2.2.3 Polymorphic 8
2.2.4 Metamorphic 9
2.2.5 General obfuscation techniques 9
2.3 Malware Analysis 10
2.3.1 Static methods 10
2.3.2 Dynamic methods 11
2.3.3 Malware features for Machine Learning 11
3 Related work **13**
3.1 Opcodes as indicator for malware 13
3.2 The influence of packers 14
4 Methods **15**

4.1 Choice of methods . . . 15
4.2 Data collection . . . 15
4.3 Data preprocessing . . . 17
4.4 Reverse Engineering . . . 17
4.4.1 Disassembly . . . 17
4.4.2 Debugging . . . 18
4.4.3 x86/x64 architecture . . . 18
4.5 Machine Learning . . . 20
4.5.1 Feature selection . . . 21
4.5.2 Classifiers . . . 22
4.6 Data analysis . . . 24
5 Experiments, results and discussion . . . 27
5.1 Experimental environment . . . 27
5.1.1 System . . . 27
5.1.2 Dataset . . . 28
5.2 Experimental design . . . 29
5.2.1 Empirical analysis . . . 29
5.2.2 Computational analysis . . . 32
5.3 Results . . . 37
5.3.1 Reliability of opcodes . . . 37
5.3.2 Influence of packers . . . 37
5.3.3 N-gram lengths . . . 38
5.4 Discussion . . . 39
5.4.1 Methodology . . . 39
5.4.2 Datasets . . . 39
5.4.3 Robustness . . . 40
5.4.4 Source code and computational complexity . . . 40
5.4.5 Limitations . . . 40
6 Conclusion and further work . . . 42
6.1 Theoretical implications . . . 42
6.2 Practical considerations . . . 42
6.3 Further research . . . 43
Bibliography . . . 45
A Computational results . . . 50
B Datasets . . . 54
B.1 Asm file example . . . 54
B.2 N-gram example . . . 57
B.3 Distribution example . . . 59
B.4 Arff file example . . . 60
C Source code . . . 62
C.1 createArff4gram . . . 62
C.2 copyAllExe . . . 65

C.3 createBatAllBenign 65
C.4 getInstructionsFromAsm 66

List of Figures

1 Obfuscation arms race [1] 7
2 A simplified polymorphic engine [1] 8
3 PE-file format [2]. 16
4 Data movement 20
5 Support Vector Machine [3] 23
6 ROC curve [4]. 26
7 Datasets 29
8 Top 18 opcodes. 31
9 Experimental Design 33
10 ROC curve 1-gram all - RF from Weka. 35
11 ROC curve 2-gram SymmetricalUncert - RF from Weka. 35
12 ROC curve 3-gram SymmetricalUncert - RF from Weka. 36
13 ROC curve 4-gram SymmetricalUncert - RF from Weka. 36
14 Influence of packers on the classification accuracy 38
15 Classification accuracy using different sequence lengths 39
16 Classifier accuracies for 1-gram 52
17 Classifier accuracies for 2-gram 52
18 Classifier accuracies for 3-gram 53
19 Classifier accuracies for 4-gram 53

List of Tables

1 Opcode n-grams 2
2 Top 18 instructions found in the files 30
3 Top 15 opcodes used exclusively used by malware 31
4 Top rated features 34
5 Number of features 35
6 Accuracy for Symmetrical Uncertainty 36

1 Introduction

In this chapter the purpose and idea behind the project is presented, and an introduction of the subject is given. We state the current problem and the research questions that will be attempted answered. The scope, contributions and limitations of the thesis is discussed. Finally, an outline of the different chapters is provided.

1.1 Keywords

Malware detection, machine learning, opcode sequences, assembly instructions, data mining.

1.2 Topic covered by the project

Malware is used by both criminals and governments for a range of purposes. This can be to steal information, surveillance activity, sabotage, or remote control computers in a botnet. According to Symantec, 1 in 566 web sites scanned by their Web-site Vulnerability Assessment Service in 2013 contained malware [5]. As malware authors have started to create more sophisticated software, the identification of these malicious samples have become more difficult. Various obfuscation techniques like the use of packers, encryption and poly-/metamorphic code are huge challenges for those trying to understand what the code does.

The analysis of malware is usually divided in static and dynamic approaches. In addition there is a content based method. Previous master theses at HiG have taken all approaches. Flaglien used a static approach [6], Sand a dynamic approach [7] and Berg a pseudo-dynamic approach [8]. Borg used a content based approach [9]. This project will take a static approach, but with the help of automation.

1.3 Problem description

To identify malware, most antivirus scanners use a combination of signature matching and heuristic-based detection. The obvious problem with this is that only known and partially known samples based on the artifacts extracted by malware analysis will be recognized. Because of the rapid growth in malware samples over the last years, this is no longer sufficient.

In this thesis we use reverse engineering to extract the assembly instructions from a given file. Further, different machine learning approaches are used to look at the sequences of these instructions. The file is classified as either malware or benign depending on the sequences.

1.4 Justification, motivation and benefits

As malware is a big threat to not only individuals, but also critical infrastructure, businesses and governments, it is important to develop new and efficient ways to detect the malicious content. Some malware variants are now so advanced that they can even cause physical damage in an industrial infrastructure seemingly isolated from the online world [10].

1.5 Research questions

To understand if it is possible to develop a method to identify malware based on the opcode sequences in a file, the goal of the thesis is to answer these three questions:

1. What is the reliability of opcodes as an indicator for malware?
2. What is the influence of packers in regards to malware identification using opcodes?
3. Which n-gram lengths provide best accuracy?

As stated in the research questions, n-grams are used as representation for the data. N-grams are created by a sliding window *n* characters long across a document and recording the unique strings found [11]. In example, the 2-grams for the text 'virus' are 'vi', 'ir', 'ru', and 'us'. When it comes to malware classification, members of the IBM TJ Watson Research Center were the first to introduce the use of n-grams [11]. In table 1 an example of opcode n-grams is provided.

Opcodes	**2-gram**	**3-gram**
mov add push add	mov, add add, push push, add	mov, add, push add, push, add

Table 1: Opcode n-grams

1.6 Scope and Contributions

Since this is a project with a relatively short time frame, some limitations have to be set. IDA Pro [1] has an option to extract de-obfuscated code from memory to get a more accurate opcode representation of the packed malware [12][p. 542]. This will not be done, since the method proposed will focus on a static approach. Further, it has been shown that packed Windows system files may be classified as malware [13]. We are aware of this, but will not take this into consideration in this work. In relation to this, a question is raised in regard to whether the file classified gets the 'malware label' because it is packed or because it is malware. Again, this is not taken into consideration.

According to Moskovitch et al, the ratio in real-life scenarios is approximately 10 % malware and 90 % benign files [14]. They state that it is unclear what the percentage split should be in the training set, so we will keep it close to 50 / 50 like previous work. Since this is an experimental setup, little time will be spent on optimizing the code developed for multi thread processing, thus we will not focus on the computational efficiency.

In this thesis we seek to include packed malware in the dataset, since this is not yet done [15]. By utilizing unpackers in the feature extraction phase, we believe this is feasible. Also, the length of the n-grams will be increased, and different machine learning algorithms than previous work utilized to test the reliability of the extracted features.

[1] `https://www.hex-rays.com/products/ida/`

1.7 Thesis outline

The thesis is divided into six chapters. By using a top down approach, the idea is to introduce the reader to the different disciplines involved, before explaining how they are used in the research.

Next is a short outline for the following chapters.

- Chapter 2 states what malware is, and why it is a relevant research area. Different types of malware and the methods used to obfuscate them are presented. At last different malware detection techniques are discussed.
- Chapter 3 provides an overview of literature related to the thesis' research questions. First work on opcodes as an indicator for malware is presented, then papers on the influence of packers.
- Chapter 4 describes the methods used to conduct the thesis research. Data collection, data preprocessing, reverse engineering and machine learning are described. Further the methods for data analysis are described.
- Chapter 5 focuses on how the experiments were performed, and the results achieved. The experimental environment, the experimental design and the results are presented. After this, a discussion on the results follows.
- Chapter 6 concludes the thesis by discussing theoretical implications and practical considerations. At last we suggest possibilities for further work.

2 Malware identification

According to Symantec's last threat report, it was detected far more malware in 2014 than in previous years [5]. There were more than 317 million new pieces of malware created last year, meaning nearly one million new threats were released into the wild each day.

In this chapter we state what malware is, and why it is a relevant research area. Different types of malware are explained, and obfuscation techniques described. Further the different methods for malware analysis are introduced.

2.1 Different types of malware

Malware is short for malicious software. Various definitions exists, in example this from Skoudis et al [16]:

> Malware is a set of instructions that run on your computer and make your system do something that an attacker wants it to do.

Based on this definition, it does not have to be executables like we are using in this thesis. It doesn't even have to be software, as it may be implemented in hardware. The second part of the definition may refer to a wide variety of scenarios. An attacker may simply want to cause harm, in example deleting a lot of valuable files on the system. Or, the goal may be money, so the files are encrypted and the victim is asked to pay for the decryption key. Further, the reason for an attack might be espionage or theft of information like credit card numbers.

A similar and more recent definition is provided by Srakew et al:

> Malware is malicious code or software which can be vulnerable to the system in many aspects.

Unlike the earliest types of malware that were created as pranks or for vandalism, today's malware is very different. Now malware is part of a big underground economy, and is a tool used by underground organizations to earn money, and by governments for espionage and attacks [11].

Several types of malware exists. To make it easier to share information, malware should be categorized. This would also make it easier to "clean up" after a security breach in example at a company. If the malware found was a rootkit different procedures must be followed than if it was a worm. Unfortunately no true industry standard exists [17]. The Computer Antivirus Researcher's Organization (CARO) have developed a naming standard for malware, but this only serves as a general guide. Vendors are not required to follow the standard which contains the subcategories virus, dropper, trojan, PWS (Password stealer) and backdoor.

Microsoft have their own list which is longer and more detailed. It consists of the following [18]: Adware, Backdoor, Behavior, BrowserModifier, Constructor, DDoS, Dialer, DoS, Exploit, HackTool, Joke, Misleading, MonitoringTool, Program, PWS, Ransom, RemoteAccess, Rogue, SettingsModifier, SoftwareBundler, Spammer, Spoofer, Spyware, Tool, Trojan, TrojanClicker, TrojanDownloader, TrojanDropper, TrojanNotifier, TrojanProxy, TrojanSpy, VirTool, Virus and Worm.

As a higher level of distinction, malware can be divided into two main categories: Parasitic malware that needs a host program, like virus and backdoors, and self-contained programs that can operate independently like worms and bots [19][p. 216].

Next follows a description of some of the most used malware types.

2.1.1 Virus

The term virus in regards to computers was first introduced by Cohen in 1987 [20]. A virus is a piece of software that can infect other programs by modifying them [19][p. 220]. They attach themselves to other programs and execute in the background while the host program does what it is supposed to do. It consists of three parts: The infection mechanism, the trigger and a payload. The first is the way the virus "reproduces" or spread. The second is the condition on which the virus activates or delivers its payload. The last is the malicious activity it performs.

Several types of viruses exists. In addition to classify it by targets, like boot sector, file and macro, we can differentiate on how the virus tries to hide itself [19][p. 224]:

- Encrypted virus: With this type of virus, the virus itself encrypts the rest of its code with a random key. When it duplicates it uses a different key, so that no constant pattern can be observed by investigators.
- Stealth virus: The main key is that it tries to hide itself from detection. It can in example be by being the same length as a benign program. By interrupting the I/O routines, it may detect when someone is reading the portion of the disk used by itself, and then present itself as the original uninfected program.
- Polymorphic virus: By changing its bit patterns, the virus will create different signatures for every version. Like the stealth virus, the purpose is to avoid detection.
- Metamorphic virus: Same as the polymorphic, but both the behaviour and appearance is changed. This makes it even more difficult to detect the new version.

2.1.2 Worm

Stallings et al states that "a worm is a program that can replicate itself and send copies from computer to computer across network connections. Upon arrival the worm may be activated to replicate and propagate again" [19][p. 231]. This is typically done in one of two ways: By exploiting vulnerabilities in a network service, or by email [11]. Although without any sources to back up the statement, it would be safe to assume that different social media sites now serves as a third option.

Much like a virus, the worm consists of three phases [19][p. 231]:

1. Search for other systems to infect.
2. Establish a connection with the remote system.
3. Copy itself to the remote system, and make the copy run on the new system.

In addition it may try to figure out if the new system is already infected before infecting it itself.

2.1.3 Bot

A bot is a program that secretly takes control over a computer. It is short for robot, and was originally called remote access trojan horse [21]. A computer that is infected is also called a bot, or a zombie [22]. Attackers usually aim to get hundreds or thousands computers infected at the same time. This way all the infected computers can be controlled and used in a coordinated manner. This is called a botnet [19][p. 240].

The botnets are controlled using Command and Control servers (C&C). The communication can go over different protocols. Some use IRC, both via channels and private messages, some use HTTP and interpret the response messages as commands, while some use peer-to-peer (p2p) based communication [22].

There are several reasons for a malware author to use bots. The most common are distributed denial-of-service (DDoS) attacks, spamming, traffic sniffing, keylogging, spreading new malware and installing adware [19][p. 240-241]. In the recent years botnets have also been used for bitcoin mining [23].

2.1.4 Rootkit

A rootkit is a set of programs that gives the attacker administrator access to the system, and at the same time hides its presence. The name is originally from the administrator account *root* in Unix/Linux, and a set of tools that provided this level of access. These included *ps*, *netstat*, *ls* and *passwd* [24].

Due to the administrator privileges, rootkits can be very difficult to detect. They can intercept calls to APIs and alter the responses. This way the process monitor, file lists and registers may display wrong information [19][p. 242].

2.1.5 Backdoor

A backdoor is a secret way into a program that bypasses the normal security procedures. This way it allows unauthorized access into the system. It can be triggered by a special series of input or being run with a special user id [19][p. 216].

2.1.6 Trojan horse

A trojan horse is something that appears useful or harmless, but does more than what it pretends to. Typical examples are fake antivirus programs that pretend to scan for malware, but instead does something else in the background [11]. It got its name from the Trojan horse in Greek mythology.

It should be mentioned that these types of malware are often combined. In example a trojan horse may be used to install a rootkit on the attacked system.

2.2 Obfuscation techniques

To make the programs harder to understand, malware authors use different obfuscation techniques. This is done to avoid being detected by antivisrus programs, and to make it more difficult for reverse engineers to understand what the program is doing [25]. It is also used by legit companies and developers to protect their intellectual property (source code). An obfuscated program will do the same as the original code, but the new code will be more difficult to understand.

Since malware authors strive to make it more difficult to detect their malware, antivirus creators on the other hand, have had to follow up with new detection methods [1]. Figure 1 shows a simplified overview of the arms race.

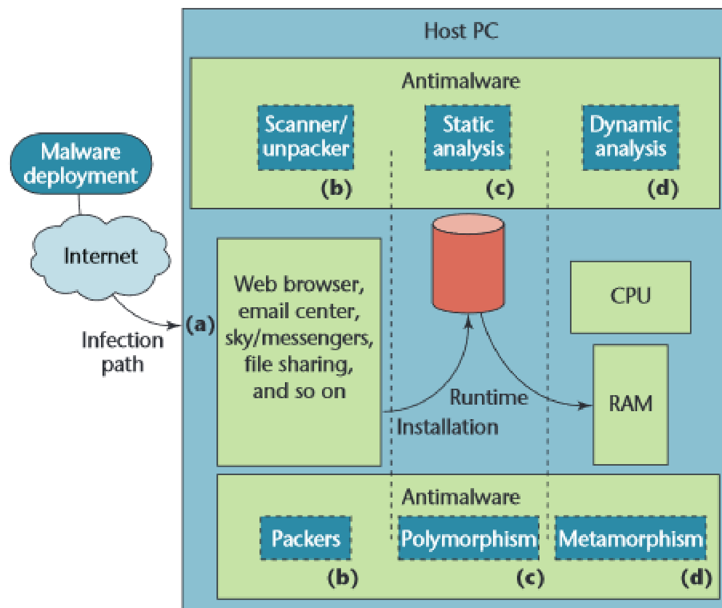

Figure 1: Obfuscation arms race [1]

Next we present different methods used to obfuscate malware.

2.2.1 Encryption

To hide the malicious part of the code, this part is often encrypted [26]. As stated previously, a new, random key is used for every infection. This way the signature of the file will be different every time. The encrypted part of the code will be decrypted only when the file is run.

The drawback with this method is that the part of the code that performs the decryption will be the same in all versions. Because of this, the virus can still be recognised based on the pattern of the decryption code [26].

2.2.2 Packers

In addition to encryption, packers can be used to change the signature of a file and avoid detection. Despite the fact that packers were originally created to decrease the file size due to limitations on disk space and bandwidth, they are now frequently used by malware authors [1]. Packers are known as good obfuscation tools because a small change in one of the files being packed will result in a very different signature. It is also easy to pack the same malware with several different packers, again to get different signatures [1].

2.2.3 Polymorphic

Polymorphism goes a step further. Instead of changing the runtime code like packers, the static binary code is changed [1]. Again, the purpose is to change the signature of the file.

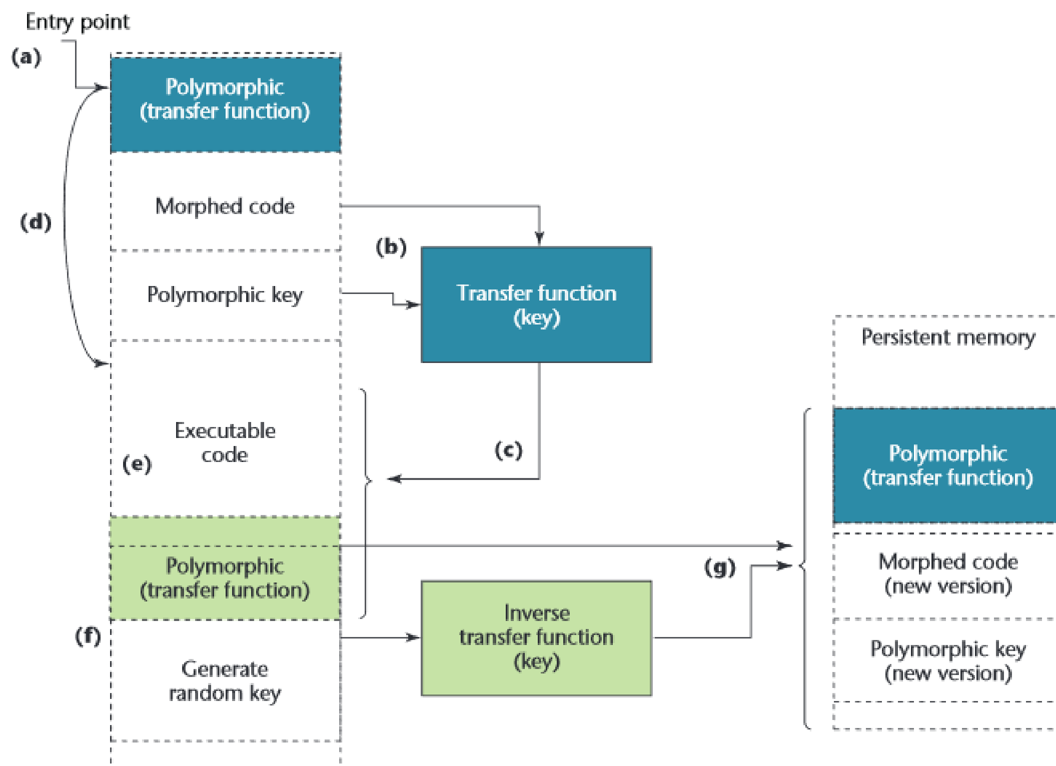

Figure 2: A simplified polymorphic engine [1]

Figure 2, taken from OKane et al [1] shows a simplified polymorphic engine.

(a) First the malware gets it turn on the processor.

(b) The mutated malware is deciphered into machine instructions.

(c) These machine instructions is written to memory.

(d) The deciphered malware is run.

(e) The malicious activity is done.

(f) A new key is generated.

(g) The machine instructions are transformed back with the new key.

Note that the machine instructions loaded in memory are the same for every version of the malware. As a result of this, the signature may be recognised when running the code in a sandbox [26].

2.2.4 Metamorphic

While the machine instructions in memory are the same in every version of mutated malware using a polymorphic approach, this is not the case with the metamorphic approach. With this approach, the instructions loaded in memory is changed, and this is written back to the infected file [1].

2.2.5 General obfuscation techniques

The following obfuscation techniques are often used in the previously described polymorphic and metamorphic categories of malware.

String obfuscation

String obfuscation is used to make it more difficult to understand the content of the variables in the code. It may be achieved by splitting the strings into smaller parts or by encoding them.

Name randomization

The variables themselves and the function names can also be obfuscated. By using randomized names, it does not give away information on what it is used for, thus it have to be understood from the context which require a higher skill level by the examiner.

Dead code insertion

Dead code insertion means to add code that do not change the abilities of the program. The code is just there to confuse anyone looking into it, or to change the signature of the file. It can be simple instructions that does nothing like the *nop* instruction, or it can be a series of instructions that ends up at the starting point. Some authors also add complex code that is never executed [27].

Register reassignment

With register reassignment, the registers used is switched in the new version of the malware [28]. In example the *EAX* register can be reassigned to the *EBX* register. It does not introduce any time delay [29].

Subroutine reordering

By changing the order of the subroutines in the code, the signature of the file is easily changed. With *n* subroutines, it can be generated *n!* different variants of the file [28].

Instruction substitution

Yet another technique is instruction substitution. The idea is to use different instructions which give the same results. An example is to substitute *mov* with *push* and *pop* [28].

Code transposition

By changing the order of the sequences, further confusion can be created. One can either shuffle the instructions and use jumps to get the correct execution order, or one can analyse which instructions are dependent on each other, and switch the ones that are not [28].

Anti debugging

To complicate the debugging process different anti debugging techniques can be used. Branco et al [30] provides a thorough review of several possible techniques. This is outside the scope of this thesis, but for completeness we list a few of them in this section.

- PEB NtGlobalFlag
  The presence of a value in this field might indicate a debugger.
- IsDebuggerPresent
  A kernel32 function that uses the BeingDebugged field in PEB (Process Environment Block) to detect a debugger.
- Heap Flags
  Several heap flags indicate the presence of a debugger.
- Self debugging
  The process can create a copy of itself and attach the copy to itself as a debugger. This way a real debugger can not attach to it, since only one debugger can be attached to a process at once.

2.3 Malware Analysis

In this section different methods for identifying malware is presented. There are two main approaches for analysing malware: static and dynamic [31]. During static analysis the malware sample is not run, while during dynamic it is.

2.3.1 Static methods

Static analysis is usually performed first when dealing with an unknown file. The first step is to scan the file manually with the installed antivirus program on the host [32]. If the file is already known, there is no point in spending hours trying to figure it out by yourself. (Except for the learning experience.) In addition to the systems antivirus program, the file can be ran through a site like *VirusTotal* which scans the file using 43 different antivirus programs. It may also be useful to calculate the *hash* of the file, and search for it on-line to see if anyone else has encountered the same file.

String analysis is a simple way to get hints about the file. By listing all strings in the file information like command line options, user dialogue, passwords, URLs e-mail addresses, libraries and function calls can be found [32].

Disassembly is a vital part of the static analysis. By retrieving the assembly instructions from a binary, one can investigate the source code to understand what the program does. To understand it though, one will have to have deep knowledge in the (in our case) x86 and x86-64 architecture and Windows internals. Disassembly is discussed further in section 4.4.1. To make the code easier

to understand, it can be decompiled as well. This way the code is represented in a higher-level language. Although the code will not be just like the original, it will ease the task for the one examining the code.

2.3.2 Dynamic methods

The simplest way to perform dynamic analysis is to run the sample and monitor what happens. It is important to only run it in an isolated environment like a sandbox, or an off-line lab. The dynamic approach should always be taken *after* static analysis has been performed. When executing a malware sample, there are several aspects that should be monitored [32][p. 597-612]:

- File activity
  Malware may read files to gather information, start other programs or load DLLs. To alter other programs, files may be written or changed. A good tool to record all activity on the file system is *Diskmon* [33].

- Processes
  To register processes, *Process Explorer* [34] can be used. With this tool, all files, registry keys and DLLs the process has loaded are logged. Also, the processes are organized in a tree-structure, so it is easy to see if the process has spawned any new processes.

- Network activity
  Since a lot of malware use the network connection to receive commands and/or send information, the network activity should be monitored. *TCPView* [35] is a tool to investigate which ports are listening for incoming traffic. To gather all information sent and received through the network Wireshark [36] can be used.

- Registry access
  The registry on Windows is a database containing configuration keys for the operating system and several of the programs installed. Changing a registry key can have huge impact on the security on the system. Again, *Process Monitor* can be used to monitor the register changes.

Debugging is another way to perform dynamic analysis. This method is explained in section 4.4.2.

2.3.3 Malware features for Machine Learning

Executable files contain several characteristics that can be used in combination with machine learning to perform classifying. They can be grouped in five categories [11]. The first is binary based features. This is features that can be obtained from the binary file directly. In this group we have n-grams of the byte code, information from the PE-header, and the previous mentioned strings.

The second category is disassembly based features. In this category opcodes, which are used in this work, and operands, which are the remainder of the instruction are found.

The third group is control flow based features. The control flow of a program is the order of which the different parts of the code is executed. Subgroups in this category are call graphs

and control flow graphs. The difference between them is that the former represents the relationship between the procedures, while the latter represents the control flow within one of these procedures.

The fourth category is semantic based features. This kind of features focus on what the code is doing rather than the code itself. Subcategories are state change, which is about changes in the registry and memory, and API calls which are the calls a program executes to use a provided library.

The fifth and last is hybrid features. To increase the performance of a classifier several of the previous features or feature groups are combined.

3 Related work

This chapter contains an overview of related literature that is already published. The focus is on the literature related to the research questions, so dynamic approaches are not mentioned. The first section presents work related to the first and the third questions, while the second section is on the second question.

3.1 Opcodes as indicator for malware

To deal with unknown malware which classic signature-based methods can not handle, there have been developed two different approaches: anomaly detectors and data mining-based detectors [15]. Anomaly detectors create a profile based on benign software, and when a file deviates from the profile it is flagged as suspicious. Data mining-based looks at characteristics from both datasets and classifies a file based on these characteristics.

In 2005, Li et al [37] proposed to use 1-gram representation of normalised byte value distribution of a file to identify the file type. This proved to be highly accurate on both exe, gif, jpg, pdf and doc files with an average accuracy of 98.9 % when using the K-means algorithm. Yu et al have done similar experiments [28].

From Bilar [38], we know that the opcode distribution is different in malware and benign software. On a data set of 67 malicious and 20 benign samples, about one third of the opcodes had the same frequency, one third higher, and one third lower in malware versus benign. Also, malware contains a higher rate of rare opcodes.

Moskovitch et al [39] conducted an experiment using byte sequence n-grams on a data set with more than 30,000 files. They took into account the imbalance problem: that there are significantly more instances of one class compared to another. With only 15 % malicious files in the dataset they achieved an accuracy of 99 %. The Weka implementations of artificial neural networks, decision trees and naive bayes were used. To reduce the complexity of the n-grams, term frequency was used to select only the top 1,000 byte codes. By doing this, n-grams of up to n=6 could be used. Interestingly n=2 gave the best results, possibly due to the small number of malware used compared to benign files.

By comparing malware detection through assembly and Application Programming Interface (API) call sequences Shankarapani et al [29] discovered that opcodes had a higher accuracy, but was more computationally expensive. They also found that packers were used by both classes of files, while encryption was only used by malware.

Santos et al [15] used the following method: NewBasic Assembler was used to disassemble the files. Then, an opcode profile was generated. This was a list of how many times the different opcodes were used in the malicious and benign dataset. Further, the opcode relevance was computed. This was done by using mutual information to measure the statistical dependence between the variables. The opcodes were grouped in n-grams of lenght n=1 and n=2. The classification algorithms used were decision trees, support vector machines, k-nearest neighbours

and Bayesian networks. Support vector machines with a normalised polynomial kernel, and n-gram length of n=2 gave the best result (95.9 %).

The most recent study we have found was conducted by Zolothukin et al [40]. They used a clustering algorithm based on iterative support vector machines to identify malware. N-grams of length n=1 and n=2 were used. An accuracy of 97 % was achieved with n=2 and ReliefF as a dimensionality reduction method.

3.2 The influence of packers

A packer is a software program that compress and encrypt other executable files and restore the original file when the packed file is loaded into memory [2].

Several researchers have published papers on detecting whether a packer has been used on a specific file. However, it seems to be little published research on detecting whether a packed file is malware or benign.

Jacob et al [41] presented a way to determine if a given malware sample is similar to a previous seen example, even if the new file was packed. This is done without unpacking the code, thus operating directly on the packed files. Their approach is also capable of determining if a file is unpacked, compressed, encrypted or multi-layered encrypted.

With PE-Probe, Shafiq et al [42] extracts features from the headers of all major portions of a PE file. The file is first classified as packed or unpacked by feeding the info to a multi-layer perceptron. If the file is classified as packed, a subset of features that are robust to packing are compared with the features of the file.

4 Methods

This section describes the methods that will be used to conduct the research. The research questions and the way of evaluating the answers are discussed. Further the data collection and data preprocessing is presented. We propose how to obtain the datasets, and what have to be done in order to use them correctly. After this, reverse engineering and machine learning is introduced. Both disciplines are central for the thesis. The feature selection methods and classifiers to be used are explained. At last the data analysis will be discussed. This includes both empirical analysis, and computational analysis.

4.1 Choice of methods

Research in general is characterized by quantitative and qualitative approaches [43]. While quantitative research focus on numbers, descriptive statistics, figures and illustrations to show results of the study, qualitative research deal with descriptions of concepts and perceptions mainly by interpretations. To answer the three research questions, quantitative methods will be used.

- Question 1: What is the reliability of opcodes as an indicator for malware?

When answering this question, it is natural to measure the accuracy and false positive rate of the classification. The numbers will be compared to the results of previous studies, like Santos et al [15].

- Question 2: What is the influence of packers in regards to malware identification using opcodes?

Again, accuracy and false positive rate will be measured. The experiments will be ran both with and without the use of unpackers, and the results compared. Examples of unpackers are PolyUnpack [44], Renovo [45] and OmniUnpack [46].

- Question 3: Which n-gram lengths provide best accuracy?

The results from the different n-gram lengths using each of the different machine learning algorithms and feature selections will be compared.

4.2 Data collection

It is important to use high quality and representative data sets when conducting research. Executable files exists for all three main operating systems. These include Portable Executable files (PE-files) on Windows, Executable and Linkable Format (ELF-files) on Linux, and Mach Object (Mach-O) on Mac [2]. Since Windows is the most used, and the most attacked system [39], we decided to focus on the files on this system.

PE-files are used not only for executables, but also libraries and drivers. The filename extensions are .cpl, .exe, .dll, .ocx, .sys, .scr, .drv, .efi, and .fon. The file consists of several parts. First

the MS-DOS header, then the prefix "PE" and the PE-header. This contains information on how many sections the file consists of, the machine type and a time stamp. Next is the optional header with "most of the meaningful information about the executable image, such as initial stack size, program entry point location, preferred base address, operating system version, section alignment information, and so forth" [47]. Next is the section table header where information on each section is stored. This information includes raw size, virtual size and name for all the sections. Last is the section data. Here the *original entry point* (OEP) is located. The OEP dictates where the execution of the file begins. A figure of the PE-file format can be seen in Figure 3.

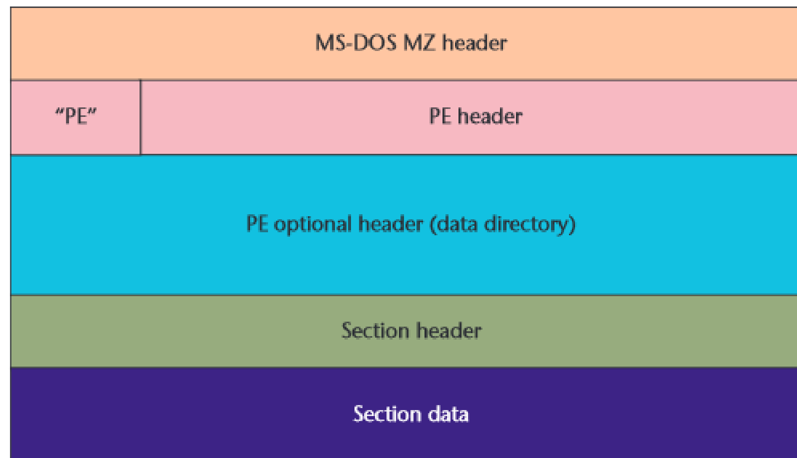

Figure 3: PE-file format [2].

There are two ways of collecting malware: setting up your own honey-pot, or downloading the files from online collections provided by others. Several such sites exist, like VX Heaven [48], PacketStorm [49] and VirusShare [50]. In this project, the malware samples will be downloaded from VirusShare [1]. The reasons for this is because gathering with a honey-pot will acquire too much time, and second, from VirusShare the samples are already grouped in collections. This makes it easier to get a relatively large collection of x86 and x86-64 PE-files.

When it comes to the benign files, the authors have not come across any available dataset online. Previous work have mainly used *dll* and *exe* system files from Windows XP [38, 39]. Santos included "text processors, drawing tools, windows games, internet browsers and PDF viewers" as well [15]. To create our dataset, we will use a combination of exe files from a clean installation of Windows Vista, and a set of popular programs. The set of programs is acquired using Ninite [51], which is an automated installer that can install a range of programs at once. Our set will include web browsers, messaging apps, media players, runtime environments, image processors, document readers, file sharing software, developer tools, online storage, compression software and others.

[1] http://www.http://virusshare.com/

4.3 Data preprocessing

To acquire the data we are interested in, we need to perform preprocessing on the PE files. All the files have to be disassembled to convert them to *asm* files. These files contains the assembly code for the PE file. To do the disassembling, a combination of IDA Pro [52] and automation by scripting with Python is used. For every file, a file with the same name but with the ending .asm is created. Further, these files are stripped for everything but the opcodes. To make sure only valid opcodes are kept, they are compared to a reference list based on Intel and AMD's processor manuals [53]. After this, the n-gram files are created. One file per n-gram per PE-file is created. Hopefully longer n-grams than previously used will be created, depending on the processing time.

To use the data with Weka, *arff files* [2] have to be created. Instead of using all possible combinations of opcodes in the n-grams (which is a very large number, e.g. 4-gram = 225.360.027.841), just the combinations present in the complete dataset is used.

A PE-file can in our case be represented as a vector. Like Santos et al, we define a program *p* as a series of instructions *I*. We then have $P = (I_1, I_2, I_3, ..., I_{n-1}, I_n)$ where *n* is the number of instructions in the file. Since we only care about the opcodes, and the sequences of them, in our case the representation of a program becomes $P = (S_1, S_2, S_3, ..., S_{n-1}, S_n)$ where *S* is a sequence of instructions *I*. *n* in this case is the number of sequences in the file.

For the *arff files*, this means the following: Each unique sequence present in the datasets becomes a feature. For every file, it is counted how many occurrences there are for every such sequence. The vector representation of a file then becomes the number of occurrences for every sequence.

4.4 Reverse Engineering

To understand what a specific file of malicious software does, reverse engineering is used. Chikofsky et al defines reverse engineering in the following way [54]:

> Reverse engineering is the process of analysing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.

Although the term was originally used in the context of analysing hardware to be able to copy it [55], it is now widely used in regards to software as well. Within the area of software, which is the field of this thesis, reverse engineering can be divided in two subgroups: disassembly and decompiling. A disassembler gives you the assembly instructions of a program, while decompiling tries to recreate the program in a higher level language, in example C.

4.4.1 Disassembly

Programming languages is usually divided in different generations [12][p. 4]:

- First generation languages.
  This is the machine language, consisting of only ones and zeroes, or hexadecimal. It is very difficult for humans to interpret, as data and instructions is very similar. It is also referred to as binary code.

[2] `https://weka.wikispaces.com/ARFF`

- Second generation languages.
  The next level is the assembly languages. At this level, the program is represented as opcodes and operands. A disassembler is used to get from the lower level binary to assembly language.

- Third generation languages.
  This is the languages normally used by programmers. C and Java are common examples. Easier abstractions like keywords and constructs are available for the programmers. A decompiler may be used to reach this level during reverse engineering.

The literature differs slightly in the definition of opcodes. In this work, we define it as the part of the instruction that is to be performed. In example in the instruction *pop ebp*, *pop* is the opcode. In example Santos et al use this definition [15], while Sikorski et al defines it as the byte-code version of the same [31].

To disassembly a binary, there are several tools available. *Dumpbin*, *objdump* and *otool* can be used to produce assembly from binaries [12]. Most debuggers as well will have the same capabilities. Debuggers are listed in the next section.

The disassemble process itself can be divided in four steps [12]:

1. Identify the entry points to the parts of the file that contains the code to be disassembled.

2. Read the values and map the binary to assembly language.

3. Output the desired syntax, usually AT&T or Intel.

4. Repeat the process for the remaining code.

4.4.2 Debugging

While disassembling provides the assembly code of a program, a debugger lets you execute a program instruction by instruction [56][p. 395]. This way it is possible to get full control of the registers, memory and stack content. It is also possible to execute only the parts of the program you need to inspect. The last part is useful to bypass anti-debugging parts of the code.

To start debugging, the analyst has mainly two options: To attach debugging to a running process or to open a process in the debugger. The benefits with the former method is that the initial actions can observed, and the process stops when the debugger is closed. With the latter, the debugging can be stopped without killing the process being examined [56].

On Windows, debuggers include *SoftICE*, *WinDbg*, *IDA Pro*, *OllyDbg* and *Immunity Debugger* [57]. On Linux *gdb*, *lldb*, *emacs*, *ddd*, *strace*, *ltrace*, *xtrace*, *valgrind* and *NLKD* are popular tools [57].

4.4.3 x86/x64 architecture

To understand basic assembly code there are a few aspects one must be familiar with. These are registers, data types, the instruction set and Windows fundamentals [25]. This thesis will not go into depth on these aspects, but we will provide a very limited introduction to the first three of them.

Registers

There are eight general purpose registers (GPRs) on the architecture. We list the registers and what they are normally used for:

- EAX - Arithmetic operations.
- EBX - Data pointer.
- ECX - Counter in loops.
- EDX - Source in string/memory operations.
- EDI - Destination in string/memory operations.
- ESI - Pointer to source in stream operations.
- EBP - Base frame pointer. This points to frames within the stack. Frames store data for functions.
- ESP - Stack pointer. This points to the top of the process stack.

All of these can be further divided. In example EAX -> AX -> AH and AL. In addition to the GPRs, there are EIP (Extended Instruction pointer) which points to points to the memory address of the next instruction to be executed, and the EFLAGS which stores the status of memory operations and other execution states.

Data types

The common data types are

- Bytes - 8 bits, in example stored in AL, BL and CL.
- Word - 16 bits, in example stored in AX, BX and CX.
- Double word - 32 bits, in example stored in EAX, EBX and ECX.

Quad words may also be used. They are created by combining two registers to get 64 bit.

Instruction Set

Data can be moved and stored in five ways. It can be stored immediately to register, immediately to memory, moved from register to register, moved between register and memory, and moved from memory to memory. When moving data, the syntax consists of an opcode, the destination and a source operand. See figure 4.

The arithmetic operations are performed using

- ADD – adds a given value.
- SUB – subtracts a given value.
- INC – adds 1.
- DEC – subtracts 1.

Opcode destination, source operand
Example:

mov esi, 0F003Fh	Set ESI = 0xF003

Figure 4: Data movement

and a set of logical instructions:

- AND – ands a given value.
- OR – ors a given value.
- XOR – xors a given value.
- NOT – reverses the bits in a given value.

The stack should also be mentioned. The stack is a *last-in first-out* data structure which supports *push* and *pop*. *Push* puts something on the top of the stack and *pop* removes something from the top of the stack. It is a contiguous memory region pointed to by *ESP*, and it grows downwards.

When it comes to control flow, high level constructs like *if/else*, *switch/case* and *while/for* are implemented through

- CMP – Compares two operands by subtracting one from the other
- TEST – Compares two operands by using AND between them
- JMP – Updates ESP with a given address
- JCC – A collection of jump commands
- EFLAGS

4.5 Machine Learning

Another central field in regards to the thesis, is machine learning. Machine learning is a subfield of artificial intelligence, and there have been made huge advances within the field during the last two decades [58]. The main objective of machine learning is to learn from data to be able to classify objects into different categories or classes. The outcome of the learning results in "rules, functions, relations, equation systems, probability distributions and other knowledge representations like decision rules, decision trees and regression trees" [58].

The learning process is usually divided in two main groups: supervised and unsupervised. In supervised learning, the target labels are provided in a training set. A subgroup of the supervised group is classification. The point of a classifier is to label an object based on a set of different features. The label can be either exactly one class, or a set of possible classes. In our case, the problem is binary, so based on the features the file is classified as either malware or benign.

When it comes to unsupervised learning, the target label is not provided. The most popular unsupervised method is clustering [58]. In this method the objects are grouped into several clusters. The number of cluster may or may not be known in advance. To perform the clustering, the similarity of the features in the different objects are taken into account.

There is no golden rule that states which classifier is the best. Quite the opposite, the *No Free Lunch theorem* states that "no classifier is to be preferred over another when no information of the problem is known" [59][p. 563]. Further, for every classifier that solves a problem well, a new problem can be created on which the classifier is useless.

Machine learning is now heavily used in the field of malware detection. It is a good fit since malware contains specific patterns and similarities because of code reuse and similar functions [11]. We will now present the feature selection methods and classifiers used in this project. The methods are chosen based on several reasons. Some of them are chosen to easily compare our work to previous, some are suggested as future work by related papers, and some are chosen because they have shown good results in other types of malware detection.

4.5.1 Feature selection

For the classifier to perform well, a high quality set of features have to be selected. The *ugly duckling theorem* states [59][p. 526]:

> Given that we use a infinite set of predicates that enables us to distinguish any two patterns under consideration, the number of predicates shared by any two such patterns is constant and independent of the choice of those patterns. Furthermore, if pattern similarity is based on the total number of predicates shared by two patterns, then any two patterns are equally similar.

The point is that it is no use with a thousand features if they do not distinguish between the classes. The process of removing irrelevant and redundant features from the data is called feature selection.

To select the features to use, there are two general methods: filter methods and wrapper methods. A third is a combination of the two, called embedded models [60]. Filter methods are the fastest, and consider statistical characteristics of the data. The feature are ranked, and the top features are selected based on either a predefined number of desired features, or everyone above a certain threshold is chosen. Wrapper methods are slower and use a classifier to choose the optimal features. In this thesis we utilize five different feature selection models, which are CFS, ChiSquared, InfoGain, ReliefF and SymmetricalUncert. We will now present the basic of each of them.

- Correlation-based Feature Selection (CFS)
  This is a wrapper model which assumes that "good feature sets contain features that are highly correlated with the class, yet uncorrelated with each other" [61]. For discrete features the correlation is based on normalised Information Gain.

- ChiSquared
  The ChiSquared attribute evaluator is a filter method that uses a statistical approach. The worth of a feature is is evaluated by computing the value of the chi-squared statistic with respect to the class [62].

- InfoGain
  Information Gain is another ranker method. It uses entropy to measure impurity in a group of objects [63].

- ReliefF
  This method is an extension of the statistical filter method Relief. The idea of the method is to rank a feature on how well its value distinguishes among instances that are near each other [40].

- SymmetricalUncert
  According to the Weka documentation this feature selection "evaluates the worth of a set attributes by measuring the symmetrical uncertainty with respect to another set of attributes" [64].

4.5.2 Classifiers

After a relevant set of features have been selected, the classifiers can be used. Especially the binary classifiers are applicable since we have two classes, "malware" and "benign". In this section we introduce the different machine learning methods used in the experiments.

- Bagging
  Bagging is a method to reduce variance. With *n* learning samples, several training sets are created by choosing *n* samples randomly from the pool. This way some of the samples may be selected multiple times, or no times at all [58]. It is used in combination with other classifiers, in our case Decision Trees. Further, a Decision Tree is a special kind of decision rules where the internal nodes are attributes, and the leaf nodes are class labels. The edges are subsets of attribute values.

- Random Forest
  A Random Forest consists as the name suggests of several decision trees. Usually at least 100 [58]. All trees use only a small subset of the available features. Each tree contributes with votes on which class the sample is. This makes up a probability distribution, and the class with highest probability is chosen.

- C4.5
  C4.5 is a previously mentioned decision tree. What is special about it is that it utilizes Information Gain (see previous subsection) to split the training data. The J48 implementation in Weka used in our experiments use a pruned decision tree [65]. This means that the tree is reduced in size when expected classification error in the sub tree is larger than the expected classification error in the current node.

- Naive Bayes
  With a Naive Bayes classifier, conditional independence of the features is assumed [58]. The classifier is based on Bayes Rule combined with decision rules.

- Bayes Net
  A Bayes Net, or Bayesian belief network, can be thought of as a Directed Acyclic Graph

(DAG) that represents the dependencies between the features [59][p. 133]. It can be implemented with several different search algorithms. In our case, the search method used is hill climbing. This is very fast, but does not guarantee the optimal solution.

- Support Vector Machines (SVM)
  Support Vector Machines are among the most accurate approaches to discriminant function classification [58]. They are successful on both regression and classification problems. The generalization is good, as the classifier is nearly as good with test samples as with training samples. The general idea is to maximize the margin. The margin is the distance between the support vectors in the two classes. Support vectors are the training examples closest to a hyperplane that separates the two classes. The hyperplane can be described as $w*x+b=0$ where w is the weight vector, normal to the hyperplane, x is the feature vector, and b is the bias.

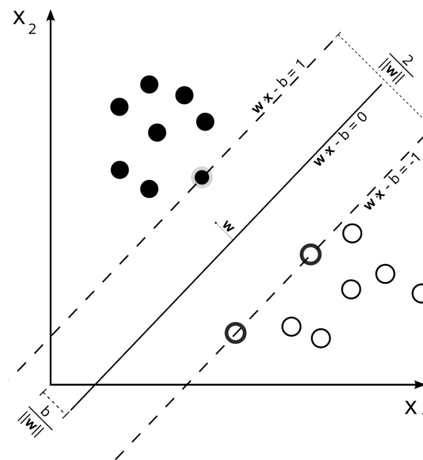

Figure 5: Support Vector Machine [3]

When the data is not linearly separable, the kernel trick is used. The idea is to represent the feature vector in a higher dimension. In the higher dimensional space, the vectors become linearly separable. For a multi class problem, the problem is divided into several sub problems. Each class is separated from the rest. To get good results, it's important to use the correct kernel function, and set the correct values to the kernel parameters. I.e. in Gaussian kernel, the sigma must be set correctly, and in polynomial kernel, the degree must be set right.

- Artificial Neural Networks (ANN)
  Artificial neural networks (ANN) are simplified mathematical models based on the human brain [66]. The usual ANN consists of several neurons. These neurons can be divided in three groups: input, hidden and output [67]. The neurons are connected by weighted connections, and they produce an output based on the weighted sum of the input signals,

and a given transfer function. ANNs have to be trained, where the training is achieved by adjusting the connection weights.

- K-Nearest Neighbours (KNN)
  This method stores all the training examples, and looks at the distance between the object to be classified and the nearest training examples. The number of neighbours, k to be taken into consideration is determined by the user. In datasets with some noise, it will help to increase the k, as the wrong samples will not count as much [58]. In our experiments we use a number of k ranging from one to ten.

- Self Organizing Map (SOM)
  A SOM can be viewed as a way of dimensionality reduction. A set of high dimensional input vectors are represented in a (typically) two-dimensional map. The map consists of several nodes into which the input vectors are mapped. Each node also contain a weight-vector, which in the case of neuro-fuzzy can be viewed as a rule.

  The SOM training algorithm is as follows [68]:

  1. Each node's weights are initialized.
  2. A vector is chosen at random from the set of training data and presented to the lattice.
  3. Every node is examined to calculate which one's weights are most like the input vector. The winning node is commonly known as the Best Matching Unit (BMU).
  4. The radius of the neighbourhood of the BMU is now calculated. This is a value that starts large, typically set to the 'radius' of the lattice, but diminishes each time-step. Any nodes found within this radius are deemed to be inside the BMU's neighbourhood.
  5. Each neighbouring node's (the nodes found in step 4) weights are adjusted to make them more like the input vector. The closer a node is to the BMU, the more its weights get altered.
  6. Repeat step 2 for N iterations.

  When it comes to the map size, several a general rule is that the map size should be $S = 5 * \sqrt{N}$ were N is the number of features in the data [69].

4.6 Data analysis

The data analysis will consist of two parts. After the creation of the n-grams, we will first look at the statistics of the data. Areas of interest will be the following:

- How many of the files in the different datasets are packed?
- What are the most frequent opcodes used?
- How many features are in the datasets?
- Is it any difference in opcodes used between the datasets?

Next, the computational experiments will be run, and the results analysed. Since we have a binary classification problem (malware vs benign software), the following performance measurements will be used: sensitivity, specificity, accuracy and ROC curve. They are defined as [58][p. 70-75]:

- Sensitivity:
  The relative frequency of correctly classified positive examples.

  $$Sens = \frac{true\ class1}{true\ class1 + false\ class2}$$

- Specificity:
  The relative frequency of correctly classified negative examples.

  $$Spec = \frac{true\ class2}{true\ class2 + false\ class1}$$

- Accuracy:
  The relative frequency of correct classifications.

  $$Acc = \frac{true\ class1 + true\ class2}{all\ classified}$$

- Receiver Operating Curve (ROC curve):
  Shows the trade off between sensitivity and specificity. The area under the curve is a measure for accuracy [4]. An example can be seen in Figure 6.

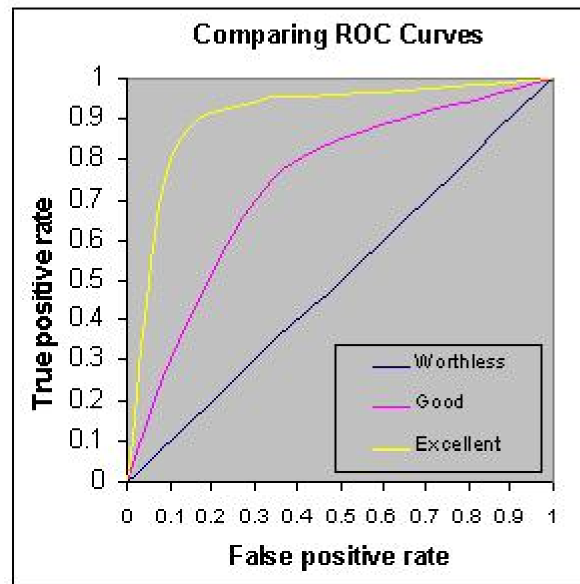

Figure 6: ROC curve [4].

In addition to the performance measures, it is interesting to evaluate the computational complexity. Although we state in the introduction that the code will not be optimized, in example to utilize multiple processor cores, the time and file size aspects are still relevant. The time spent on creating n-grams and feature selection is interesting in a online-offline point of view. To be used, the method should not be too slow. Also, the file sizes have to be reasonable.

5 Experiments, results and discussion

In this chapter the focus is on how the experiments were performed, and the results achieved. First the experimental environment is presented. Here the software, hardware and dataset is presented. This is important if someone wish to duplicate our results. Then the experimental design, including both the empirical and the computational analysis presented. At last the results are shown, and a discussion is provided.

5.1 Experimental environment

This section describes the hardware and software used, as well as the dataset obtained. Since malware is a large part of the dataset, it was given extra attention to keeping the environment sandboxed, so no hosts would be infected.

5.1.1 System

To conduct the experiments, two different machines were used. The main workstation had the following specifications:

- Processor: 6 core Intel i7 @ 3,47 GHz
- Memory: 16 GB DDR4
- Hard drive: 240 GB SSD
- Operating system: Windows 8.1

This machine was used for all tasks except the feature selection for 3- and 4-gram. Because of the extreme memory consumption due to the high number of features (up to 110 GB memory used), a virtual machine from Microsoft Azure was used for this. It had the following specifications:

- Processor: 8 core Intel Xeon E3 @ 2,4 GHz
- Memory: 112 GB DDR4
- Hard drive: 1.536 GB SSD
- Operating system: Windows Server 2012 R2

At the main workstation several virtual machines were run. To build the benign dataset, and to create the *.asm* files for that dataset, Windows Vista was used. A different Vista machine was used to create the *.asm* files for the malicious dataset.

For convenience, the rest of the work was done directly on the host operating system on the main workstation, since the files used at this point could not cause any harm.

When it comes to the software, the following scripting tools, platforms and libraries were used:

- Protection ID 6.6.7 [70]
- Python 2.7.9 [71]
- SciPy Python library [72]
- LibSVM [73]
- WEKA Classification Algorithms [74]
- Windows command line interface
- Java VM 1.7
- Weka 3.6 (Explorer and API) [75]
- VMWare Workstation 11

5.1.2 Dataset

As stated in section 4.2, the malware samples were downloaded from *VirusShare* [50]. The collection contained 992 different x86 and x64 malicious *PE-files*. According to *Protection ID* [70], 293 of them were packed. *ProtectionID* is able to detect 228 different free and commercial packers and installers [76]. The version used was last updated in December 2014.

Initially, the plan was to use unpackers to get the 'real' assembly code from the packed files. However, the unpackers in previous work [44, 45, 46] seemed not to be publicly accessible. We decided therefore to split the dataset in two: One set containing all the files (packed and not packed), and one set without the packed files.

The benign dataset, created from Windows Vista system exe files and a set of popular applications consists of 771 samples. Of them, 14 were packed. The same was done for this set, they were split in a full set, and a set without packed files.

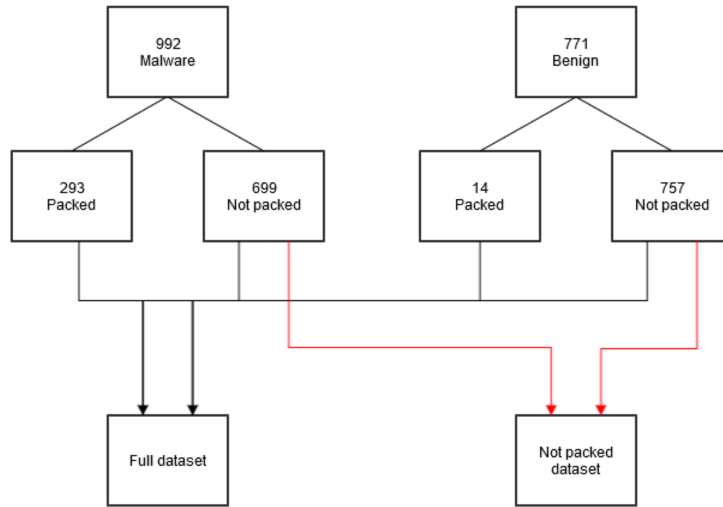

Figure 7: Datasets

5.2 Experimental design

This section explains the experimental process, and how the results were analysed. It consists of two parts: the empirical analysis and the computational analysis. The first is the initial analysis were the statistics of the raw data is examined. We show the distribution of opcodes for both malware and benign. It is shown how some opcodes are used exclusively by malware, and the most used opcodes are described. The empirical analysis helps as a guide to further run the computational experiments. In the latter, the use of feature selection and classifiers is presented. We show that most of the features selected reoccur in several of the selection methods.

5.2.1 Empirical analysis

After the creation of the n-gram files, we had some interesting data. In total there were 529 different instructions used. The top 18 of them made up 92,5 % of all the occurrences of instructions. These are shown in table 2. Of these again, *mov* accounted for 73.86 %. *Mov* was pretty equally represented in both malware and benign with 36.88 and 36.98 % of the total instructions. This is also the case in Bilar's dataset [38]. There *mov* was the most used instruction as well, with 30 % in the malware files, and 25 % in the benign. The next instructions differ in his and our work.

Opcode	Description	Malware %	Benign %
mov	Move	36,88	36,98
call	Call Procedure	8,89	8,02
lea	Load Effective Address	7,76	4,37
cmp	Compare Two Operands	5,94	5,30
push	Push Word/DW/QW Onto the Stack	2,16	8,49
jz	Jump short if zero/equal (ZF=0)	4,41	4,58
test	Logical Compare	3,27	3,97
jmp	Jump	3,59	3,06
add	Add	3,34	3,08
jnz	Jump short if not zero/not equal	2,81	2,89
pop	Pop a Value from the Stack	2,15	3,52
xor	Logical Exclusive OR	3,24	2,05
sub	Subtract	2,57	1,55
retn	Return from procedure	1,68	1,66
movzx	Move with Zero-Extend	1,33	1,06
and	Logical AND	1,13	0,95
or	Logical Inclusive OR	0,68	0,51
inc	Increment by 1	0,50	0,54

Table 2: Top 18 instructions found in the files

The malware samples contained 62 assembly instructions that were not present in the benign samples, and vice versa. Among these were several virtual machine operations. It is natural to assume that this has to do with the fact that many malware authors make their malware check whether or not their malware is being run in a sandboxed environment. If that is the case, the payload is not delivered, to avoid detection of whoever is inspecting the sample. This has previous been stated by antivirus companies like Symantec [77]. Fast system calls were also exclusively in the malware samples. The 15 most frequent of the opcodes exclusively used by malware can be seen in table 3.

All together there were 54.3 million benign and 33.9 million malware opcodes in the dataset.

Opcode	Description
stosq	Store String
syscall	Fast System Call
setno	Set Byte on Condition - not overflow (OF=0)
cvtsd2si	Convert Scalar Double-FP Value to DW Integer
movmskpd	Extract Packed Double-FP Sign Mask
prefetcht1	Prefetch Data Into Caches
fprem	Partial Remainder (for,compatibility with i8087 and i287)
cmpsq	Compare String Operands
lodsq	Load String
scasq	Scan String
cvtss2si	Convert Scalar Single-FP Value to DW Integer
fnsave	Store x87 FPU State
orpd	Bitwise Logical OR of Double-FP Values
fxsave	Save x87 FPU, MMX, XMM, and,MXCSR State
movmskps	Extract Packed Single-FP Sign Mask

Table 3: Top 15 opcodes used exclusively used by malware

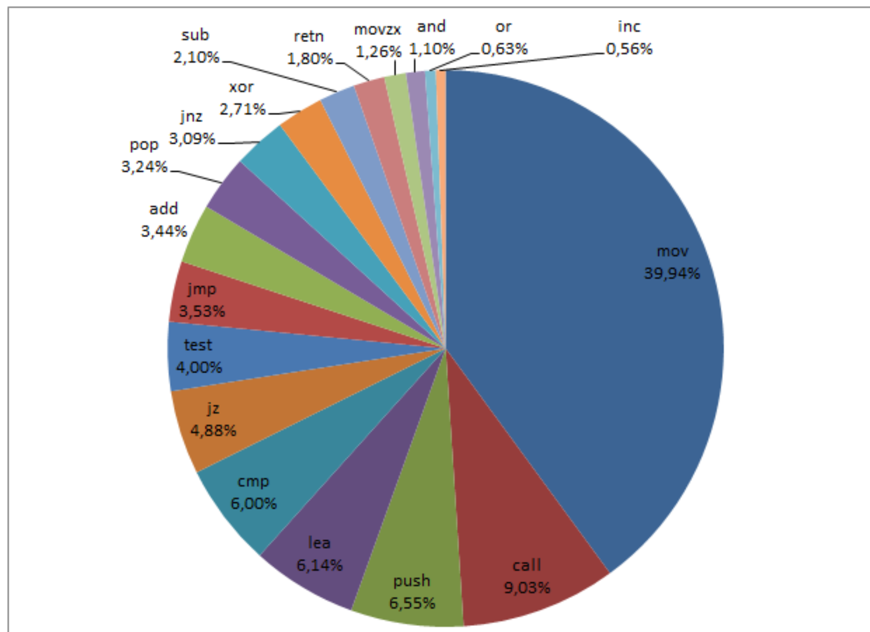

Figure 8: Top 18 opcodes.

When it comes to the 2-, 3- and 4-grams, the number of sequences rise exponentially. 2-gram contains 24,715 different sequences, 3-gram 195,993 and 4-gram 714,389. A lot of these sequences occur only one time.

- On 2-gram, the three top sequences are similar. These are ['mov', 'mov'], ['mov', 'call'] and ['call', 'mov']. After these, most of the sequences differ in order. The next two in benign is

['push', 'push'] and ['test', 'jz'], while for malware it is ['lea', 'mov'] and ['mov', 'lea'].

- On 3-gram, top four are similar. These are ['mov', 'mov', 'mov'], ['mov', 'mov', 'call'], ['mov', 'call', 'mov'] and ['call', 'mov', 'mov']. Next follows ['push', 'push', 'push'] for benign and ['lea', 'mov', 'mov'] for malware.

- On 4-gram, four of the top five sequences are the same, but not in the same order. Benign has ['mov', 'mov', 'mov', 'mov'], ['mov', 'mov', 'mov', 'call'], ['mov', 'mov', 'call', 'mov'], ['mov', 'call', 'mov', 'mov'] and ['call', 'mov', 'mov', 'mov']. Malware has ['mov', 'mov', 'mov', 'mov'], ['mov', 'mov', 'call', 'mov'], ['mov', 'mov', 'mov', 'call'], ['mov', 'call', 'mov', 'mov'] and ['lea', 'mov', 'mov', 'mov'].

On 2-gram there were 7,067 sequences exclusively used by malware. On 3-gram the number was 71,355, and on 4-gram 229,988.

5.2.2 Computational analysis

After the empirical study of the data, the automated experiments were conducted. First the *.arff* files were created. This is the file type required by the Weka collection, and consists of the name and type of all features, as well as an entry for every file and the number of the different n-grams present in that specific file.

Further the feature selection was performed, both to decrease the computational complexity and to increase the performance, as stated in the *ugly duckling theorem*. Several methods were used, as explained in section 4.5.1. The following parameters were chosen for the different feature selection methods:

- For Correlation-based Feature Subset Selection (CFS), the BestFirst search method was used. This method uses greedy hill climbing, which always selects the neighbour with the higher fitness. It is fast, but has no guarantee the local maximum found is the global maximum as well. The default values of 5 non-improving nodes to consider before search termination was used. This method is very slow and uses a lot of memory. It was only used on 1- and 2-gram since it used more than the maximum amount of memory available on longer sequences.

- For the ChiSquared method, the Ranker search method was used. This method ranks the attributes individually. On 1- and 2-gram a threshold of 0 was used. On longer n-grams, it was in addition set a maximum number of attributes (8,000).

- The Ranker method was used with the same options on the InfoGain, ReliefF and SymmetricalUsert methods as well. Since the feature selection with ReliefF took too long (more than 4 days, not completed) on 4-grams, the method was skipped on this value of n.

In addition, the classification was run for the whole attribute set with n-gram lengths of one and two. This was done to make sure that no performance was lost during the feature selection, and to see if the *ugly duckling theorem* (section 4.5.1) applied to our dataset.

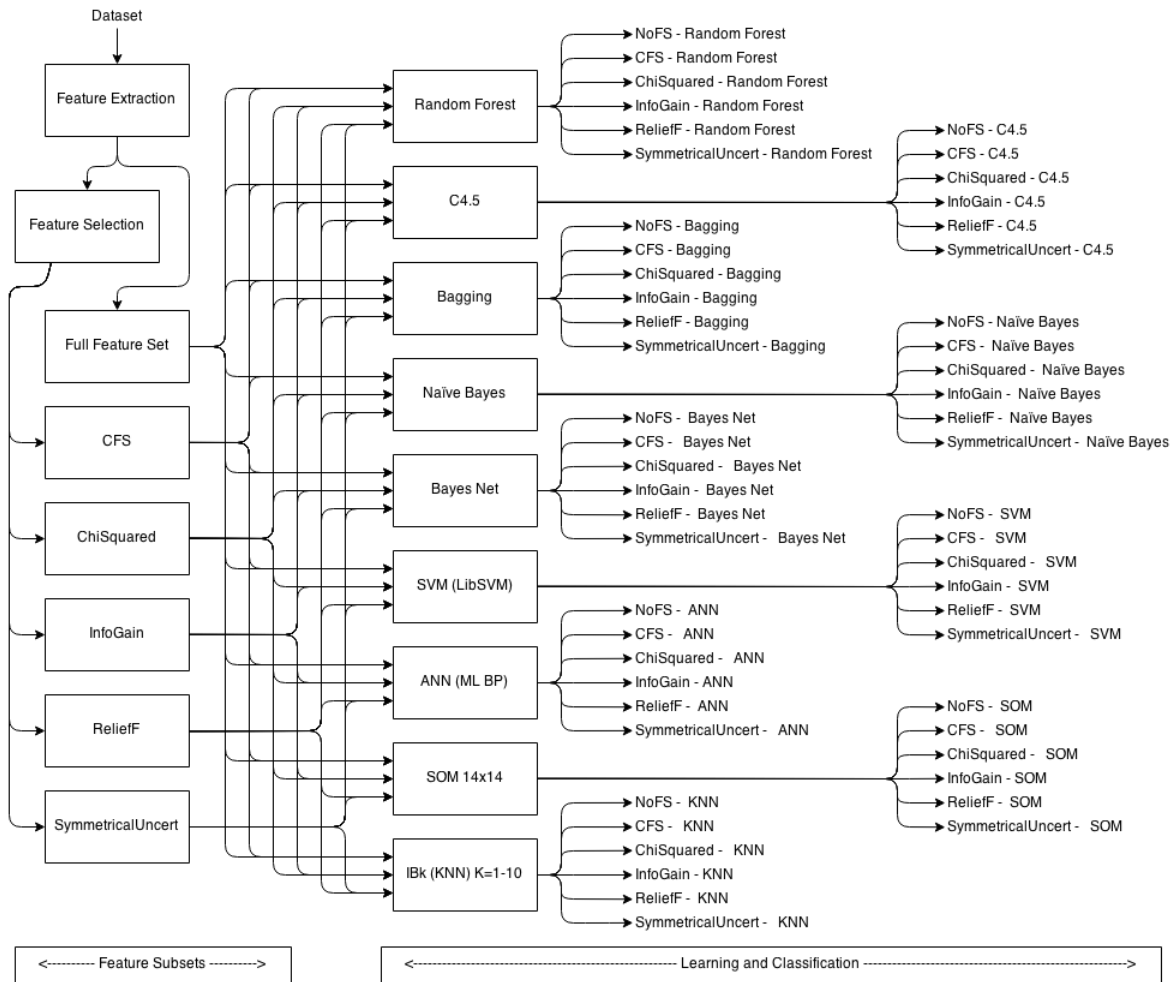

Figure 9: Experimental Design

The feature selection was the most time consuming part of the experiments. This was due to the large number of features in the full feature sets. As examples, the *arff file* for 1-gram was 1.90 MB, 2-gram was 84.1 MB, 3-gram was 666 MB and 4-gram was 2.37 GB in size. Since the feature selection methods work differently, the time spent by each of them also differed. While ChiSquared and InfoGain were fast - using only 30 minutes on 4-gram, CFS and ReliefF were very slow. ReliefF used two days on 3-gram. SymmetricalUncert was in between the two groups. It is important to remember that Weka in its normal form only utilizes one processor core, so the computing power was not utilized as well as it could have been.

All the classifiers in section 4.5.2 was used for n-grams with lengths 1 to 4. The full experimental design can be seen in figure 9.

For the Self Organizing Map, a map size of 14x14 was chosen. This was because of the rule

of thumb which states that the map size should be $S = 5 * \sqrt{N}$ were N is the number of features in the data [69].

For our method to be worth anything, it at least had to beat the ZeroR classifer. This is a classifier that labels every sample with the label of the class with most instances in it. In our case, there were more samples in the malware class, so all samples would be labelled as malware. The accuracy of this classifier is 56.27 %.

After performing attribute selection, it is interesting to see if the attributes selected by the different selection methods reoccur. All the 19 attributes from CFS on 1-gram reoccur in ChiSquared, InfoGain, ReliefF and SymmetrivalUcert. These attributes are listed in table 4, and can be considered the most significant features in the datasets.

movsxd
nop
movnti
cmovb
bt
bts
cmova
in
ror
repe
movups
leave
stosd
cld
prefetcht1
movsw
stosw
pushf
fldz

Table 4: Top rated features

The same goes for 1-gram not packed, though with slightly different attributes. On 2-gram all it is also the same, with the exception of *pop-rdtsc* on ReliefF. Without having checked all of them, it should be safe to assume there is a pattern, and that the same goes for the rest of the samples.

The number of features for each n-gram and feature selection can be seen in table 5. Since a maximum of 8,000 features was defined before running the selection methods, the datasets without packed files have the same number of features as those with packed files for 3- and 4-gram.

After the feature selection, the classifiers were run with the different created datasets. To best measure the performance, k-fold cross validation was used. This is a widely used method to test classifiers when the number of learning examples is relatively low [58][p. 83]. The idea is to split the samples in k, or in our case 10, roughly equal groups. For each of the subsets the classifier

Attribute set	1-gram	1-gram NP	2-gram	2-gram NP	3-gram	4-gram
Full feature set	530	530	24716	24716	195994	714390
CFS	19	16	30	33	-	-
ChiSquared	254	238	5008	4633	8001	8001
InfoGain	254	238	5008	4633	8001	8001
ReliefF	356	346	8074	6403	8001	8001
SymmetricalUncert	254	238	5008	4633	8001	8001

Table 5: Number of features

is trained with the other nine, and tested with the tenth. The average result of all ten test runs forms the final result.

For 1-gram including packed files, the highest achieved accuracy was 94.67 %. The classifier used was Random Forest on the full feature set. The sensitivity was 94,54 %, and the specificity 94.83 %. The true positive rate was 0.961 and the false positive rate 0.071. This gives an area under ROC curve of 0.988. Previous work [15] had an accuracy of 91.43 % using the same classifier and mutual information as feature selection method on a dataset without packed files.

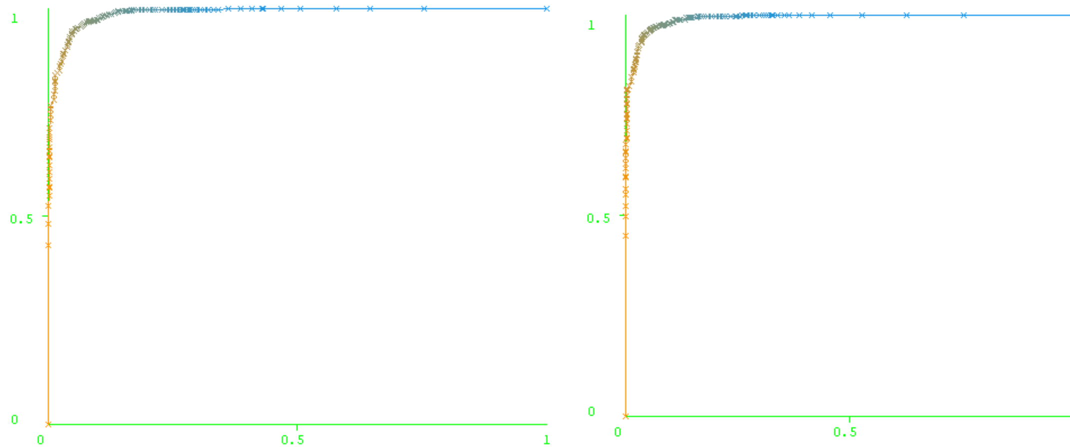

Figure 10: ROC curve 1-gram all - RF from Weka.

Figure 11: ROC curve 2-gram SymmetricalUncert - RF from Weka.

For 1-gram without the packed files, the highest accuracy was achieved with Symmetrical Uncertainty feature selection and Random Forest classifier. The accuracy was 93.06 %, the sensitivity 92.98 % and the specificity 95.03%.

For 2-gram including packed files, the highest accuracy was 95.41 %. Again the feature selection method was Symmetrical Uncertainty and the classifier Random Forest. The true positive rate was 0,968 and the false positive rate 0.064. This gives a sensitivity of 96.77% and specificity of 93.64 %. The area under ROC curve is 0.991.

For 3-gram including packed files, the highest accuracy was 95.58 % which is the highest in the performed experiment. This was also with Symmetrical Uncertainty and Random Forest.

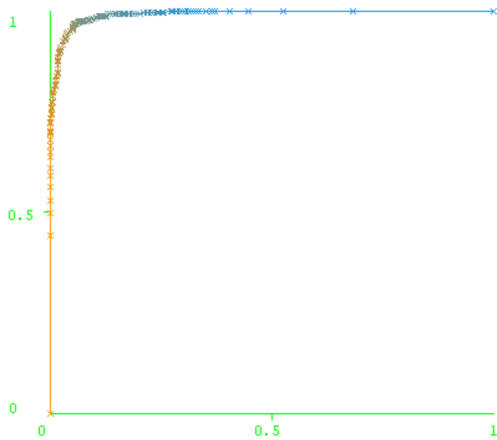

Figure 12: ROC curve 3-gram SymmetricalUncert - RF from Weka.

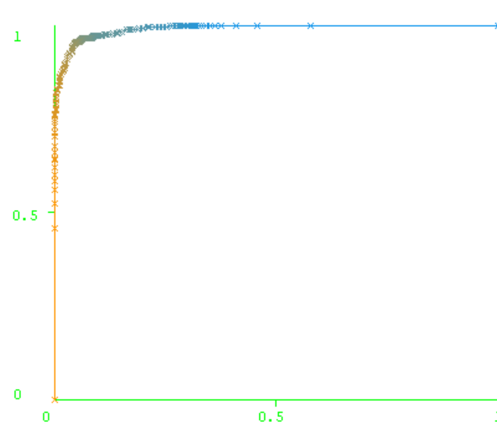

Figure 13: ROC curve 4-gram SymmetricalUncert - RF from Weka.

The true positive rate was 0.968 and the false positive rate 0.06. The sensitivity 96.77% and specificity 94.03%. The area under ROC curve is 0.991.

For 4-gram including packed, the accuracy was 95.12 %. The true positive rate 0.958 and the false positive rate 0.057. The sensitivity was 95.77 % and the specificity 94.29 %. The area under ROC curve was 0.99.

Classifier	1-gram	2-gram	3-gram	4-gram
Random Forest	94,55 %	95,41 %	95,58 %	95,12 %
J48 (C4.5)	92,57 %	93,53 %	94,61 %	93,59 %
Bagging	93,19 %	94,21 %	94,27 %	94,50 %
Naïve Bayes	63,58 %	66,59 %	70,28 %	72,26 %
Bayes Net	77,71 %	75,78 %	75,44 %	76,80 %
SVM: RBF	74,93 %	79,64 %	81,23 %	82,64 %
ANN: ML BP	1,30 %	43,39 %	34,77 %	70,79 %
SOM: 14x14	77,25 %	77,82 %	76,91 %	78,16 %
KNN: K=1	91,32 %	94,50 %	93,87 %	92,63 %
KNN: K=2	89,51 %	93,48 %	92,68 %	91,72 %
KNN: K=3	90,13 %	94,04 %	92,40 %	91,89 %
KNN: K=4	89,96 %	93,65 %	92,12 %	91,83 %
KNN: K=5	88,88 %	92,68 %	91,61 %	91,21 %
KNN: K=6	88,83 %	92,85 %	91,66 %	90,70 %
KNN: K=7	88,54 %	92,29 %	91,09 %	90,30 %
KNN: K=8	88,71 %	92,46 %	91,04 %	89,68 %
KNN: K=9	87,92 %	92,17 %	90,87 %	89,51 %
KNN: K=10	87,41 %	91,95 %	90,58 %	88,71 %

Table 6: Accuracy for Symmetrical Uncertainty

The accuracy for all classifiers, sequence lengths and datasets can be seen in the appendix.

5.3 Results

In this section we present the results of the experiments, and try to conclude in regards to the three research questions. First the reliability of opcodes in regards to malware identification is discussed in light of the results. Second, we look at the influence of packers on the results. Last, the accuracies achieved with different sequence lengths are compared.

5.3.1 Reliability of opcodes

Bilar has previously shown that there is a significant difference in the distribution of opcodes in malware and benign files. However, this was a while ago, and the evolution of obfuscation techniques has come a long way since then. In addition, the dataset used was relatively small.

In our empirical study we show that it still is a significant difference between the two classes of software. Not only is the distribution among the top opcodes different, but there are several rare opcodes used exclusively by one class.

When it comes to the computational experiments, we show that combined with the correct feature selection method and the appropriate classifier, the accuracy is high, and false positive rate low. The highest accuracy achieved with a sequence length of one was 94.67 % using the full feature set and Random Forest as algorithm. The next highest was 94.55 % using the SymmetricalUncert feature selection, also this with Random Forest.

We are aware however, that even though promising results are achieved in our experiments, there may be other factors that should be taken into account. The virus collection used was created and published in July 2013. A lot has probably happened in the malware author community since then. New zero day vulnerabilities arise, and new obfuscation techniques are created.

5.3.2 Influence of packers

The second research question was regarding the influence of packers on the results. We wanted to figure out if packed malware could be identified as well, without first performing unpacking.

To answer this, it is natural to compare the results on the dataset with packed files, to the results on the dataset without packed files.

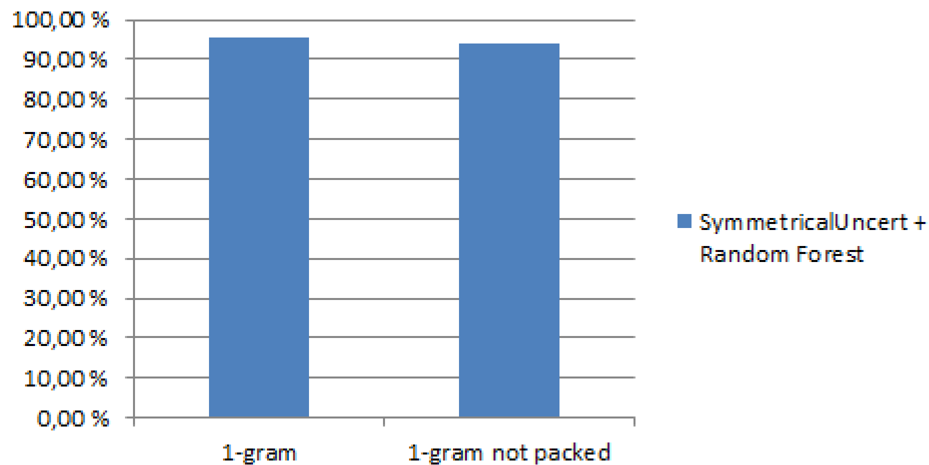

Figure 14: Influence of packers on the classification accuracy

As seen in figure 14, the difference between the two datasets is virtually non-existing. On a sequence length of one, SymmetricalUncert as feature selection method, and Random Forest as the algorithm, the accuracies were 94.55 % and 94.02 %. The dataset that included packed files performed slightly better than the one without. The difference is similar for longer sequences.

Although the influence of packers seems very small in our experiments, the results may have looked different if a larger number of packed benign files were present in the datasets. Previous work by Sarvam, has showed that 70 % of Windows system files are classified as malware by *Virustotal* if they are packed before inspection [13]. On the other hand, since the tool used to check if a file is packed is not guaranteed to detect all packers, there might be a slightly higher number of packed files.

5.3.3 N-gram lengths

The final research question was "which n-gram lengths provide best accuracy?" Since previous work had used a maximum sequence length of two, it was interesting to see if there was any correlation between sequence length and accuracy.

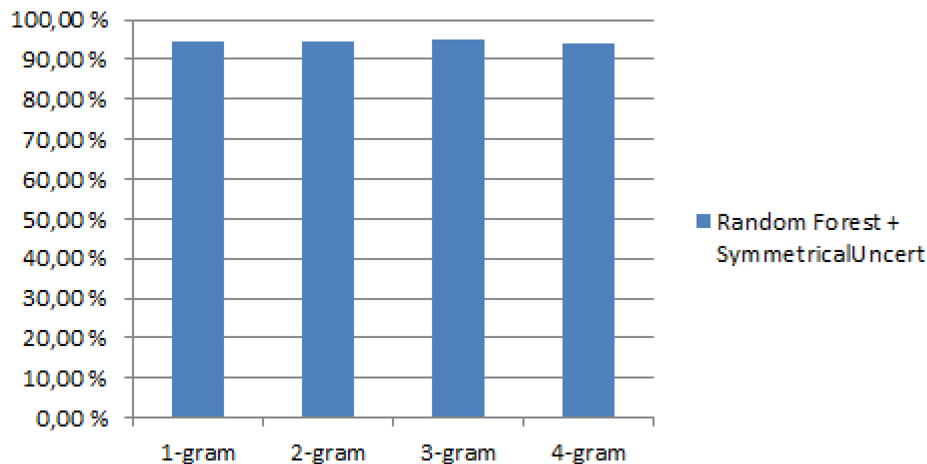

Figure 15: Classification accuracy using different sequence lengths

As seen in figure 15, the sequence length had little influence on the results. One reason for the fact that longer n-grams do not seem to be significantly more accurate, may be that longer sequences are more easily obfuscated. Seen from a forensics viewpoint though, the fact that 1- and 2-grams gives equally good results is good news. Since the computational complexity increases exponentially with longer sequences, shorter n-grams are preferred.

5.4 Discussion

This section provides a discussion on the experiments conducted, and the results achieved. The methodology, datasets and robustness of the method is discussed. At last, the limitations of the thesis are listed.

5.4.1 Methodology

The methods of using reverse engineering to extract the opcodes, and machine learning to process them was chosen to keep a static approach to the problem. By never running the program, it is made sure that the host system is not infected. As newer malware is created to detect, and some even escape a sandbox, it can be argued that a static approach is the safer way to go. The drawback is that some obfuscated code never is revealed.

When it comes to the way the performance is measured, this is the standard way for a binary classification problem. Although the main focus has been on the accuracy, the false positive rate may be just as important. It would be very frustrating for someone about to open a file they knew was benign, only to see the antivirus has flagged it as malware and quarantined it.

5.4.2 Datasets

The malicious dataset used was downloaded from *Virusshare*. To validate it, AVG antivirus was used to scan the files and confirm that they were in fact malicious. Optimally, it would be interesting to have the samples grouped by the types introduced in section 2.1, but this was not

done. For now, we only have the hash-sums and whether or not it is packed by a known packer. Another point regarding the malware dataset is that it is from July 2013. It most certainly has been some advancement in the field of malware writing since then.

The benign dataset was scanned by AVG antivirus as well, tom make sure all files were benign. The dataset consists of a very low number of packed files. To our knowledge no statistics on the number of packed benign files is available, so it might be that packers are mostly utilized by malware authors, since the file size and bandwidth is no longer an issue. A positive point on the dataset is that it consists of a lot of frequently used programs. Previous work on PE-files seem mostly to use files from the *system32* folders. An example of this is Saini et al from 2014 [78].

5.4.3 Robustness

Robustness can be seen as the methods ability to detect malware that is obfuscated. In terms of obfuscated by the use of packers, the method performs well. Previous work has achieved 95.90 % accuracy with sequence length 2 on a dataset without packed files. We show that close to the same result is achieved also with packed files.

Another definition of robustness is how well the classifier performs on new test data. If it is *overfitted* to the training data available, it may perform poor on slightly different data. That is why noise/extreme values should be filtered out during the feature selection process.

5.4.4 Source code and computational complexity

For the automatic task, scripts were written in Pyhton and Java. In addition a few *bat scripts* were used, although these were in turn generated by the Python scripts. The *bat scripts* were used to locate all exe files on the system when creating the benign dataset. They were also used to run IDA Pro in batch mode when creating the *asm files*. The creation of the n-gram files went relatively fast. The benign dataset consisted of the largest files, and the time spent on 2-gram was 11 minutes, 3-gram was 80 minutes and 4-gram was approximately 13.5 hours. As a one-time operation to create a reference dataset this is acceptable. Further, the creation of the *arff files* to be used with Weka was done in a few hours.

5.4.5 Limitations

The thesis has the following limitations:

- The developed method is tested on Windows PE-files only. Given a disassembler which is able to extract the assembly code from Linux elf files, it might work on this kind of files as well, since the underlying architecture (x86/x86-64) is the same.
- The malware dataset is almost two years old. A lot has happened the last two years, so the method should be tested on newer samples.
- All opcodes in the *asm files* are listed regardless of which procedure they are in. The result is that there will be a sequence in example of the last opcode in one procedure combined with the first opcode in the following procedure, even though they are not executed sequentially.
- The size of the Self Organizing Map is not optimized after feature selection.
- On Support Vector Machines only the default kernel for non-linear problems was used.

- Several n-gram lengths was not merged and used in combination. This might have improved the accuracy.

6 Conclusion and further work

The goal of this thesis was to contribute to the research field by advancing on the work by Santos et al in [15]. By attempting to answer three research questions, we believe this was done. In the following sections the theoretical implications, which includes the new knowledge gained is presented. Further practical considerations, which includes important information for anyone seeking to redo the work, is presented. At last, we suggest topics to focus on in further research.

6.1 Theoretical implications

In this section the theoretical implications of the research are presented. Through the work in this thesis malware was detected using a static approach. By extracting only the opcodes from PE-files, we show that sequences of these can be used for classification. From an empirical study of the extracted opcodes, we showed that n-grams are worth considering as a further research field because malware and benign files differ on the assembly level. Malware use several opcodes which are not common in benign files, and the distribution is also different.

Through the experiments it was found that it is no point in using longer sequences than 2-gram, since the accuracy does not improve noteworthy on 3- and 4-grams. As stated previously, this may be because longer sequences are more easy to obfuscate, and they may differ depending on compiler and/or packer used.

Further, our research proved that also malware obfuscated by the use of packers is recognised using this method, as long as it is present in the training set. An accuracy of 95,58 % was achieved.

As the *no free lunch* theorem states, no solution fits all problems. During the experiments the performance of the classifiers differed a lot. Different versions of decision trees: Random Forest, C4.5 and Bagging, in addition to k-nearest neighbours had the best performance on the datasets used. According to the *ugly duckling theorem* irrelevant and redundant features should be removed. After feature selection using Symmetrical Uncertainty, there were 254 features instead of 529 on 1-gram, and 5,008 instead of 24,716 on two-gram. Overall, except for 1-gram, this feature set outperformed both the full feature set and the other selection methods. *Minimum description length* (MDL) is also relevant in regards to this. We seek to find a trade off between the number of features and the performance. Even though fewer features may lead to lower accuracy, the computational complexity will decrease enough to justify a small loss.

6.2 Practical considerations

For someone who wish to repeat the experiments, the following is of interest. The malicious part of the dataset was obtained from *Virusshare*. One have to apply for membership to the site, so it may take some time before the access is granted. The benign dataset was created from Windows Vista system files and a set of popular applications. These were collected using *Ninite*.

The feature selection uses a lot of resources on the machine, especially memory. Since the

input vectors consists of a huge number of features (up to 714,390 on 4-gram) the memory usage reaches above 110 GB in some cases.

Almost none of the benign files were detected as packed, so this is something to be aware of when doing a similar experiment later. The same goes for the compiler used, as this may have an impact on the results as well.

In our experiments the same size of the self organizing maps was used for all feature subsets and sequence lengths. It may improve the results of this classifier by adjusting the size to each subset individually. As mentioned in section 4.5.2 we used the rule of thumb to determine the size. Recent work has shown improved accuracy by using a new way to determine the size [69]. On the support vector machine a *radial basis function* kernel was used. This kernel maps the input vector to a higher dimension.

A real life implementation of a system like this would probably have to be on-line. This is because of the large amount of training data, and the resources needed to create the sequences. Alternatively, the asm-file could be created locally, and the on-line part of the system could do the rest. This way the file to be transferred would be relatively small, and the resource use on the consumer's computer kept low.

6.3 Further research

In relation to this thesis, there are several areas that should be researched further. Since a master thesis only has limited time available, the following interesting topics were not investigated.

- Include packed benign
  The experiments should be run with a dataset containing a higher number of packed benign files. Our experiment only had 14 packed benign files, so there it would be interesting to see if this has any influence on the results. (Although only a bit higher than one third of the malware was identified as packed).

- Dynamic unpacking
  To get a more precise representation of malware which are obfuscated trough encryption other methods like encryption, dynamic unpacking should be used. By running the malware in a sandboxed environment and extracting the opcodes from memory, even better results might be acquired.

- Scaling
  Since malware databases are huge, some testing of the scalability of this kind of method should be performed. It is a big difference between performing feature selection on two thousand files and on several million.

- Include operands
  Even though opcodes are indicators of malware by themselves, the method could be combined with other data. API calls have proved to be effective in malware detection, so a static approach where the two are combined might be a good match.

- Opcode hexvalues
  Instead of using the opcodes as done in this work, the hexvalue of the opcodes could be

used in stead. This way there will be more crisp data, as several hexvalues maps to one opcode.

Bibliography

[1] OKane, P., Sezer, S., & McLaughlin, K. 2011. Obfuscation: The hidden malware. *IEE Security and Privacy Magazine*, 9.

[2] Yan, W., Zhang, Z., & Ansari, N. 2008. Revealing packed malware. *IEEE Security and privacy magazine*.

[3] Wikipedia. Accessed may 2015. Support vector machine. `http://en.wikipedia.org/wiki/Support_vector_machine`.

[4] Franke, K. & Chanda, S. 2011. Performance evaluation of classifiers. *Lecture notes*.

[5] Symantec. Accessed april 2015. 2014 internet security threat report, volume 19. `http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf`.

[6] Flaglien, A. O. 2010. Cross-computer malware detection in digital forensics. *HiG master thesis*.

[7] Sand, L. A. 2012. Information-based dependency matching for behavioral malware analysis. *HiG master thesis*.

[8] Berg, P. E. 2011. Behavior-based classification of botnet malware. *HiG master thesis*.

[9] Borg, K. 2013. Real time detection and analysis of pdf-files. *HiG master thesis*.

[10] Bencsath, B., Pek, G., Buttyan, L., & Felegyhazi, M. 2012. The cousins of stuxnet: Duqu, flame, and gauss. *Future Internet*.

[11] LeDoux, C. & Lakhoita, A. 2015. Malware and machine learning. *Intelligent Methods for Cyber Warfare*.

[12] Eagle, C. 2011. The ida pro book.

[13] Sarvam Blog. Accessed april 2015. Nearly 70% of packed windows system files are labeled as malware. `http://sarvamblog.blogspot.no/2013/05/nearly-70-of-packed-windows-system.html`.

[14] Moskovitch, R., Stopel, D., Feher, C., Nissim, N., Japkowicz, N., & Elovici, Y. 2009. Unknown malcode detection and the imbalance problem. *Journal in Computer Virology*.

[15] Santos, I., Brezo, F., Ugarte-Pedrero, X., & Bringas, P. 2013. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*.

[16] Skoudis, E. & Zeltser, L. 2010. Malware: Fighting malicious code.

[17] Dube, T., Raines, R., Peterson, B., Bauer, K., & Rogers, S. 2010. An investigation of malware type classification. *Proceeding of the 5th International Conference Information Warfare and Security*.

[18] Microsoft Malware Protection Center. Accessed may 2015. Naming malware. `http://www.microsoft.com/security/portal/mmpc/shared/malwarenaming.aspx`.

[19] Stallings, W. & Brown, L. 2008. Computer security principle and practise. *Pearson Education*.

[20] Cohen, F. 1987. Computer viruses: theory and experiments. *Computers and security*.

[21] McLaughlin, L. 2004. Bot software spreads, causes new worries. *IEEE Computer Society*, 5(6).

[22] Goebel, J. & Holz, T. 2007. Rishi: Identify bot contaminated hosts by irc nickname evaluation.

[23] Plohmann, D. & Gerhards-Padilla, E. 2012. Case study of the miner botnet. *International Conference on Cyber Confl ict*, 4.

[24] McAfee. 2006. Rootkits, part 1 of 3: The growing threat. *Whitepaper*.

[25] Dang, B., Gazet, A., & Bachaalany, E. 2014. Practical reverse engineering. *Wiley*.

[26] You, I. & Yim, K. 2010. Malware obfuscation techniques: A brief survey. *International Conference on Broadband, Wireless Computing Communications and Applications*.

[27] Borello, J.-M. & Mé, L. 2008. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3), 211–220.

[28] Yu, S., Zhou, S., Liu, L., & Yang, R. 2010. Malware variants identification based on byte frequency. *Second international conference on networks security, wireless communications and trusted computing*.

[29] Shankarapani, M. & Ramamoorthy, S. 2010. Malware detection using assembly and api call sequences. *Springer-Verlag France*.

[30] 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 USA*.

[31] Sikorski, M. & Honig, A. 2012. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press.

[32] Skoudis, E. 2004. *Malware: Fighting malicious code*. Prentice Hall Professional.

[33] Microsoft Technet. Accessed may 2015. Diskmon for windows v2.01. `https://technet.microsoft.com/en-us/sysinternals/bb896646`.

[34] Microsoft Technet. Accessed may 2015. Process explorer v16.05. `https://technet.microsoft.com/en-us/sysinternals/bb896653`.

[35] Microsoft Technet. Accessed may 2015. Tcpview v3.05. `https://technet.microsoft.com/en-us/sysinternals/bb897437`.

[36] Wireshark Foundation. Accessed may 2015. Wireshark homepage. `https://www.wireshark.org/`.

[37] Li, W.-J., Wang, K., Stolfo, S., & Herzog, B. 2005. Fileprints: Identifying file types by n-gram analysis. *Proceedings of the 2005 IEE Workshop on Assurance and Security*.

[38] Bilar, D. 2007. Opcodes as predictor for malware. *Int. J. Electronic Security and Digital Forensics, Vol 1, No. 2*.

[39] Moskovitch, R., Feher, C., Tzachar, N., Berger, E., Gitelman, M., Dolev, S., & Elovici, Y. 2008. Unknown malcode detection using opcode representation. *Springer-Verlag Berlin Heidelberg*.

[40] Zolotukhin, M. & Hamalainen, T. 2014. Detection of zero-day malware based on the analysis of opcode sequences. *The 11th Annual IEEE SSNC - Security, Privacy and Content Protection*.

[41] Jacob, G., Comparetti, P. M., Neugschwandtner, M., Kruegel, C., & Vigna, G. 2013. A static, packer-agnostic filter to detect similar malware samples. *Springer-Verlag Berlin Heidelberg*.

[42] Shafiq, Tabish, & Farooq. 2009. Pe-probe: Leveraging packer detection and structural information to detect malicious portable executables. *Virus Bulletin Conference*.

[43] Moghaddam, G. G. & Moballeghi, M. 2008. How do we measure use of scientific journals? a note on research methodologies. *Scientometrics*.

[44] Royal, P., Halpin, M., Dagon, D., Edmonds, R., & Lee, W. 2006. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*.

[45] Kang, M. G., Poosankam, P., & Yin, H. 2007. Renovo: A hidden code extractor for packed executables. *WORM 07*.

[46] Kang, M. G., Poosankam, P., & Yin, H. 2007. Omniunpack: Fast, generic, and safe unpacking of malware. *23rd Annual Computer Security Applications Conference*.

[47] Kath, R. 1997. The portable executable file format from top to bottom.

[48] VX Heaven. Accessed dec 2014. Vx heaven. `http://vxheaven.org/`.

[49] Packet Storm Security. Accessed dec 2014. Packet storm security. `http://packetstormsecurity.com/`.

[50] VirusShare. Accessed may 2015. Virusshare. http://virusshare.com/.

[51] Secure By Design Inc. Accessed may 2015. Ninite. http://ninite.com/.

[52] Hex-Rays. Accessed dec 2014. Ida pro. https://www.hex-rays.com/products/ida/.

[53] x86asm.net. Accessed may 2015. X86 opcode and instruction reference. http://ref.x86asm.net/coder-abc.html.

[54] Chikofsky, E. J., Cross, J. H., et al. 1990. Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7(1), 13–17.

[55] Rekoff, M. 1985. On reverse engineering. *Systems, Man and Cybernetics, IEEE Transactions on*, (2), 244–252.

[56] Ligh, M., Adair, S., Hartstein, B., & Richard, M. 2010. *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code*. Wiley Publishing.

[57] Wikibooks. Accessed may 2015. x86 disassembly/analysis tools. http://en.wikibooks.org/wiki/X86_Disassembly/Analysis_Tools.

[58] Kononenko, I. & Matjazkukar, M. 2007. Machine learning and data mining. *Woodhead Publishing*.

[59] Duda, R. O., Hart, P. E., & Stork, D. G. 2012. *Pattern classification*. John Wiley & Sons.

[60] Liu, H. & Motoda, H. 2007. *Computational methods of feature selection*. CRC Press.

[61] Hall, M. A. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

[62] Frank, E. Accessed may 2015. Weka class chisquaredattributeeval. http://weka.sourceforge.net/doc.stable/weka/attributeSelection/ChiSquaredAttributeEval.html.

[63] Shapiro, L. Accessed may 2015. Information gain. http://homes.cs.washington.edu/~shapiro/EE596/notes/InfoGain.pdf.

[64] Zhao, Z. Accessed may 2015. Weka class symmetricaluncertattributeseteval. http://weka.sourceforge.net/doc.packages/fastCorrBasedFS/weka/attributeSelection/SymmetricalUncertAttributeSetEval.html.

[65] Frank, E. Accessed may 2015. Weka class j48. http://weka.sourceforge.net/doc.dev/weka/classifiers/trees/J48.html.

[66] Fullér, R. 1995. Neural fuzzy systems.

[67] Abraham, A. 2005. Artificial neural networks. *handbook of measuring system design*.

[68] Buckland. Accessed may 2015. Kohonen's self organizing feature maps. http://ai-junkie.com.

[69] Shalaginov, A. & Franke, K. 2015. A new method for an optimal som size determination in neuro-fuzzy for the digital forensics applications. *13th International Work-Conference on Artificial Neural Networks, IWANN 2015, Palma de Mallorca, Spain*, 13(2).

[70] CDKiLLER & TippeX. Accessed may 2015. Protectionid homepage. http://pid.gamecopyworld.com/.

[71] Python Software Foundation. Accessed may 2015. Python homepage. https://www.python.org/.

[72] Jones, E., Oliphant, T., Peterson, P., et al. 2001–. SciPy: Open source scientific tools for Python. [Online; accessed 2015-05-19].

[73] Chang, C.-C. & Lin, C.-J. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2, 27:1–27:27. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm.

[74] Brownlee, J. Accessed may 2015. Weka classification algorithms. http://wekaclassalgos.sourceforge.net/.

[75] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. 2009. The weka data mining software: An update. SIGKDD Explorations. http://www.cs.waikato.ac.nz/ml/weka/.

[76] Tool Library, C. Accessed may 2015. Packer identifiers. http://www.woodmann.com/collaborative/tools/index.php/Category:Packer_Identifiers.

[77] Wueest, C. Accessed may 2015. Symantec official blog. http://www.symantec.com/connect/blogs/does-malware-still-detect-virtual-machines.

[78] Saini, A., Gandotra, E., Bansal, D., & Sofat, S. 2014. Classification of pe files using static analysis. In *Proceedings of the 7th International Conference on Security of Information and Networks*, 429. ACM.

A Computational results

This spreadsheet contains all the classification accuracies for all sequence lengths, feature subsets and classifiers. It also includes additional graphs.

Data set	Attribute Selection	Random Forest	J48 (C4.5)	Bagging	Naïve Bayes	Bayes Net	SVM (LibSVM)	ANN (ML BP)	SOM 14x14	IBk (KNN) K=1	K=2	K=3	K=4	K=5	K=6	K=7	K=8	K=9	K=10	Number of attributes
1-gram	Full feature set	94,67 %	92,57 %	93,19 %	63,81 %	77,71 %	76,06 %	45,55 %	77,14 %	91,32 %	89,56 %	90,19 %	89,68 %	88,66 %	88,60 %	88,66 %	88,77 %	88,20 %	88,09 %	530
	CFS	91,15 %	89,51 %	89,05 %	66,53 %	82,19 %	79,92 %	76,52 %	79,30 %	89,28 %	89,22 %	88,49 %	88,66 %	88,43 %	87,92 %	88,20 %	87,69 %	87,86 %	88,43 %	20
	ChiSquared	94,38 %	92,51 %	93,31 %	63,58 %	77,71 %	74,93 %	56,15 %	77,25 %	91,32 %	89,51 %	90,13 %	89,96 %	88,88 %	88,83 %	88,54 %	88,71 %	87,92 %	87,41 %	254
	InfoGain	93,36 %	92,51 %	93,36 %	63,58 %	77,71 %	74,93 %	56,15 %	77,25 %	91,32 %	89,51 %	90,13 %	89,96 %	88,88 %	88,83 %	88,54 %	88,71 %	87,92 %	87,41 %	254
	ReliefF	93,70 %	92,51 %	93,25 %	63,76 %	77,71 %	75,50 %	6,92 %	77,14 %	91,44 %	89,68 %	90,30 %	89,68 %	88,77 %	88,71 %	88,77 %	88,77 %	88,20 %	87,92 %	356
	SymmetricalUncert	94,55 %	92,57 %	93,19 %	63,58 %	77,71 %	74,93 %	1,30 %	77,25 %	91,32 %	89,51 %	90,13 %	89,96 %	88,88 %	88,83 %	88,54 %	88,71 %	87,92 %	87,41 %	254
1-gram not packed	Full feature set	93,68 %	91,28 %	92,17 %	57,07 %	83,10 %	71,63 %	43,54 %	73,15 %	89,29 %	87,77 %	88,60 %	87,64 %	87,16 %	86,33 %	85,92 %	85,71 %	85,58 %	85,30 %	530
	CFS	89,49 %	87,50 %	87,98 %	60,23 %	80,77 %	79,33 %	76,10 %	80,01 %	88,05 %	87,43 %	86,61 %	86,54 %	86,68 %	86,61 %	86,54 %	86,13 %	86,47 %	86,47 %	16
	ChiSquared	94,02 %	90,52 %	92,65 %	57,07 %	83,10 %	70,40 %	35,16 %	73,70 %	89,29 %	87,57 %	88,12 %	87,57 %	87,23 %	86,74 %	86,54 %	85,92 %	85,71 %	85,71 %	238
	InfoGain	92,51 %	90,45 %	92,58 %	57,07 %	83,10 %	70,40 %	8,24 %	73,70 %	89,29 %	87,57 %	88,12 %	87,57 %	87,23 %	86,74 %	86,54 %	85,92 %	85,71 %	85,71 %	238
	ReliefF	91,76 %	91,14 %	92,38 %	57,01 %	83,10 %	71,09 %	7,83 %	73,15 %	89,56 %	87,98 %	88,74 %	87,71 %	87,16 %	86,47 %	86,06 %	85,78 %	85,78 %	85,44 %	346
	SymmetricalUncert	94,02 %	90,52 %	92,51 %	57,07 %	83,10 %	70,40 %	51,99 %	73,70 %	89,29 %	87,57 %	88,12 %	87,57 %	87,23 %	86,74 %	86,54 %	85,92 %	85,71 %	85,71 %	238
2-gram	Full feature set	94,10 %	93,25 %	94,27 %	64,61 %	-	82,19 %	57,40 %	77,65 %	92,91 %	91,78 %	92,29 %	91,78 %	90,92 %	90,92 %	90,47 %	90,36 %	90,19 %	90,47 %	24716
	CFS	93,53 %	92,00 %	92,00 %	64,38 %	86,78 %	82,53 %	78,90 %	79,01 %	93,25 %	92,63 %	92,34 %	92,06 %	91,89 %	92,00 %	92,17 %	92,12 %	92,29 %	92,17 %	30
	ChiSquared	94,50 %	93,59 %	94,38 %	66,08 %	75,78 %	79,64 %	40,39 %	77,82 %	94,50 %	93,48 %	94,04 %	93,65 %	92,68 %	92,85 %	92,29 %	92,46 %	92,17 %	91,95 %	5008
	InfoGain	93,99 %	93,53 %	94,33 %	66,14 %	75,78 %	79,64 %	51,79 %	77,82 %	94,50 %	93,48 %	94,04 %	93,65 %	92,68 %	92,85 %	92,29 %	92,46 %	92,17 %	91,95 %	5008
	ReliefF	93,82 %	93,14 %	94,27 %	66,19 %	75,72 %	80,26 %	12,08 %	77,54 %	93,65 %	92,63 %	92,80 %	92,29 %	91,61 %	91,38 %	91,21 %	91,49 %	90,87 %	90,75 %	8074
	SymmetricalUncert	95,41 %	93,53 %	94,21 %	66,59 %	75,78 %	79,64 %	43,39 %	77,82 %	94,50 %	93,48 %	94,04 %	93,65 %	92,68 %	92,85 %	92,29 %	92,46 %	92,17 %	91,95 %	5008
2-gram not packed	Full feature set	92,99 %	92,24 %	93,75 %	57,35 %	-	78,98 %	49,24 %	74,79 %	92,38 %	90,87 %	91,35 %	91,07 %	90,11 %	90,11 %	88,87 %	89,01 %	88,53 %	88,74 %	24716
	CFS	92,65 %	91,21 %	91,69 %	58,10 %	85,37 %	79,74 %	77,54 %	76,51 %	93,06 %	91,90 %	91,21 %	91,55 %	91,14 %	90,80 %	90,73 %	90,38 %	90,38 %	90,45 %	33
	ChiSquared	93,06 %	92,72 %	93,48 %	59,82 %	80,77 %	75,55 %	51,99 %	75,00 %	93,61 %	92,38 %	92,79 %	92,38 %	91,41 %	91,55 %	91,00 %	90,87 %	90,87 %	90,52 %	4633
	InfoGain	93,75 %	92,72 %	93,41 %	59,89 %	80,77 %	75,55 %	12,77 %	75,00 %	93,61 %	92,38 %	92,79 %	92,38 %	91,41 %	91,55 %	91,00 %	90,87 %	90,87 %	90,52 %	4633
	ReliefF	93,68 %	92,51 %	93,54 %	59,13 %	80,63 %	75,82 %	47,18 %	74,38 %	93,20 %	91,55 %	92,03 %	92,03 %	90,80 %	90,66 %	89,70 %	90,04 %	89,42 %	89,35 %	6403
	SymmetricalUncert	93,68 %	92,72 %	93,48 %	60,30 %	80,77 %	75,55 %	42,93 %	75,00 %	93,61 %	92,38 %	92,79 %	92,38 %	91,41 %	91,55 %	91,00 %	90,87 %	90,87 %	90,52 %	4633
3-gram	Full feature set	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	195994
	CFS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	ChiSquared	94,55 %	94,33 %	94,16 %	69,77 %	75,38 %	81,28 %	25,75 %	76,97 %	93,70 %	92,51 %	92,46 %	91,66 %	91,44 %	90,75 %	90,75 %	90,47 %	90,30 %	90,13 %	8001
	InfoGain	94,04 %	94,50 %	94,21 %	69,77 %	75,38 %	81,23 %	62,45 %	76,86 %	93,99 %	92,46 %	92,46 %	91,95 %	91,49 %	91,21 %	90,70 %	90,41 %	90,13 %	90,07 %	8001
	ReliefF	93,59 %	93,48 %	94,21 %	78,50 %	73,28 %	80,71 %	44,36 %	73,96 %	93,87 %	92,80 %	93,48 %	92,74 %	92,00 %	91,15 %	91,09 %	90,53 %	90,07 %	89,90 %	8001
	SymmetricalUncert	95,58 %	94,61 %	94,27 %	70,28 %	75,44 %	81,23 %	34,77 %	76,91 %	93,87 %	92,68 %	92,40 %	92,12 %	91,61 %	91,66 %	91,09 %	91,04 %	90,87 %	90,58 %	8001
3-gram not packed	Full feature set	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	195994
	CFS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	ChiSquared	94,44 %	93,48 %	93,06 %	64,42 %	79,81 %	77,82 %	51,03 %	74,04 %	93,06 %	91,41 %	91,35 %	90,73 %	89,97 %	89,70 %	90,18 %	89,35 %	89,08 %	88,80 %	8001
	InfoGain	93,61 %	93,48 %	93,13 %	64,70 %	80,01 %	77,82 %	13,46 %	73,70 %	93,75 %	92,24 %	91,41 %	90,87 %	90,38 %	90,04 %	90,11 %	89,63 %	89,35 %	88,94 %	8001
	ReliefF	92,86 %	93,06 %	92,99 %	62,23 %	78,98 %	80,56 %	3,23 %	70,12 %	94,02 %	91,96 %	92,51 %	91,14 %	90,45 %	90,45 %	89,63 %	89,29 %	88,46 %	88,05 %	8001
	SymmetricalUncert	94,37 %	93,96 %	93,20 %	64,15 %	80,77 %	77,75 %	28,09 %	73,49 %	93,75 %	92,51 %	91,90 %	91,35 %	90,87 %	90,18 %	90,04 %	89,97 %	89,90 %	89,15 %	8001
4-gram	Full feature set	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	714390
	CFS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	ChiSquared	94,38 %	93,93 %	94,16 %	71,70 %	75,95 %	82,47 %	33,29 %	77,99 %	93,25 %	91,66 %	92,00 %	91,72 %	90,92 %	90,36 %	90,07 %	90,13 %	89,34 %	89,22 %	8001
	InfoGain	94,95 %	93,76 %	94,33 %	71,75 %	76,06 %	82,47 %	71,53 %	77,82 %	93,31 %	91,72 %	91,83 %	91,38 %	90,36 %	90,19 %	89,96 %	89,73 %	89,28 %	89,05 %	8001
	ReliefF	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	SymmetricalUncert	95,12 %	93,59 %	94,50 %	72,26 %	76,80 %	82,64 %	70,79 %	78,16 %	92,63 %	91,72 %	91,89 %	91,83 %	91,21 %	90,70 %	90,30 %	89,68 %	89,51 %	88,71 %	8001
4-gram not packed	Full feature set	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	714390
	CFS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	ChiSquared	94,02 %	92,38 %	93,06 %	66,35 %	80,70 %	79,81 %	61,95 %	73,90 %	93,75 %	92,38 %	92,65 %	92,24 %	91,07 %	90,66 %	90,04 %	89,49 %	89,08 %	88,87 %	8001
	InfoGain	93,48 %	92,24 %	92,93 %	66,41 %	80,77 %	78,81 %	49,24 %	71,91 %	93,89 %	92,65 %	92,58 %	92,10 %	91,48 %	90,80 %	90,87 %	90,25 %	89,97 %	89,56 %	8001
	ReliefF	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	SymmetricalUncert	93,68 %	92,10 %	93,06 %	66,62 %	82,55 %	80,01 %	55,63 %	73,21 %	93,68 %	92,17 %	92,65 %	92,58 %	91,62 %	90,80 %	90,87 %	90,11 %	90,18 %	89,35 %	8001

ZeroR (alle I største klassen) = 56.2677 %

Classifiers

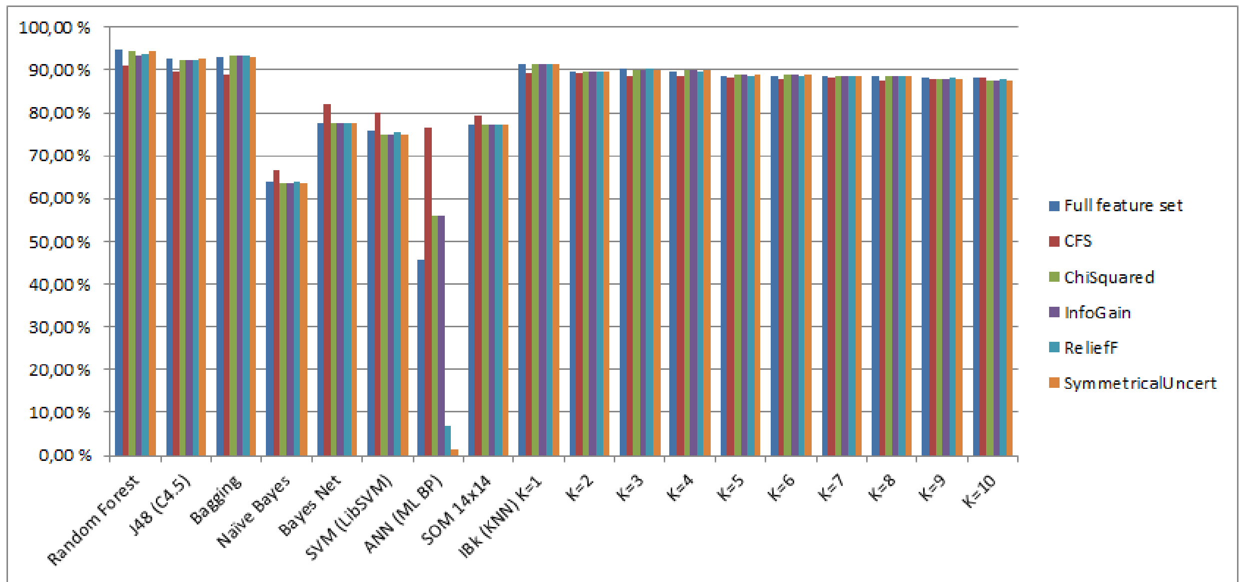

Figure 16: Classifier accuracies for 1-gram

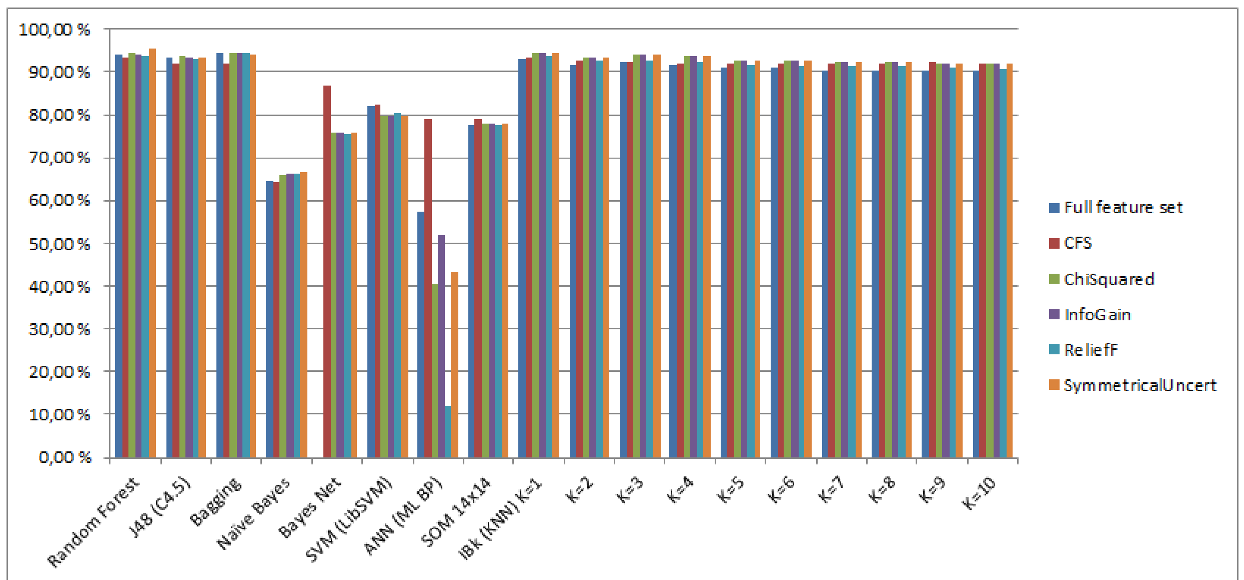

Figure 17: Classifier accuracies for 2-gram

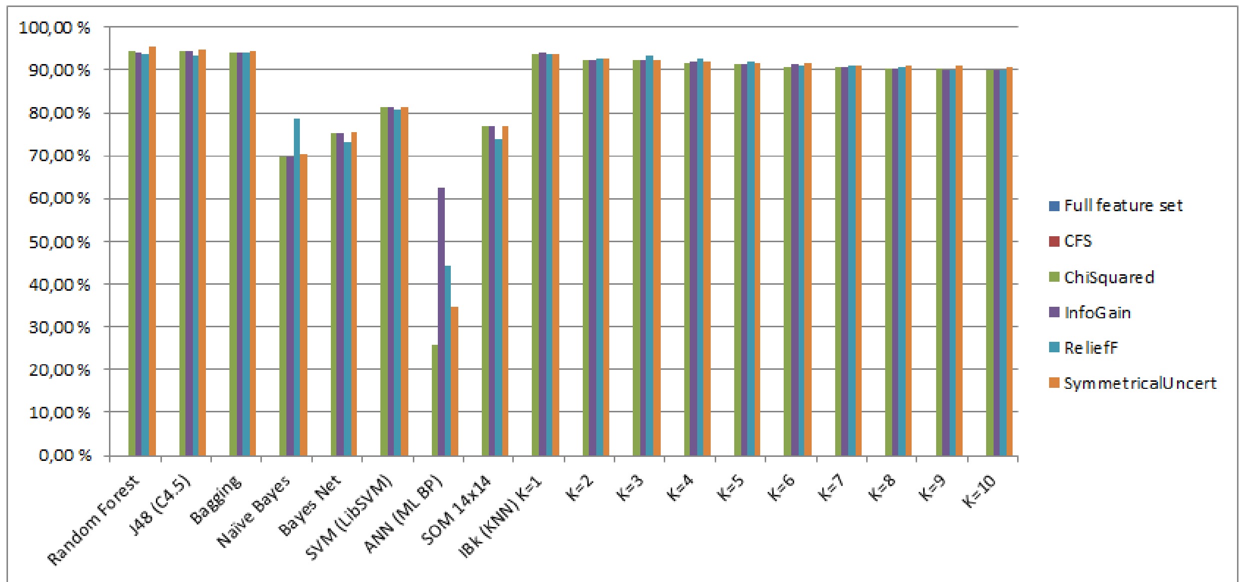

Figure 18: Classifier accuracies for 3-gram

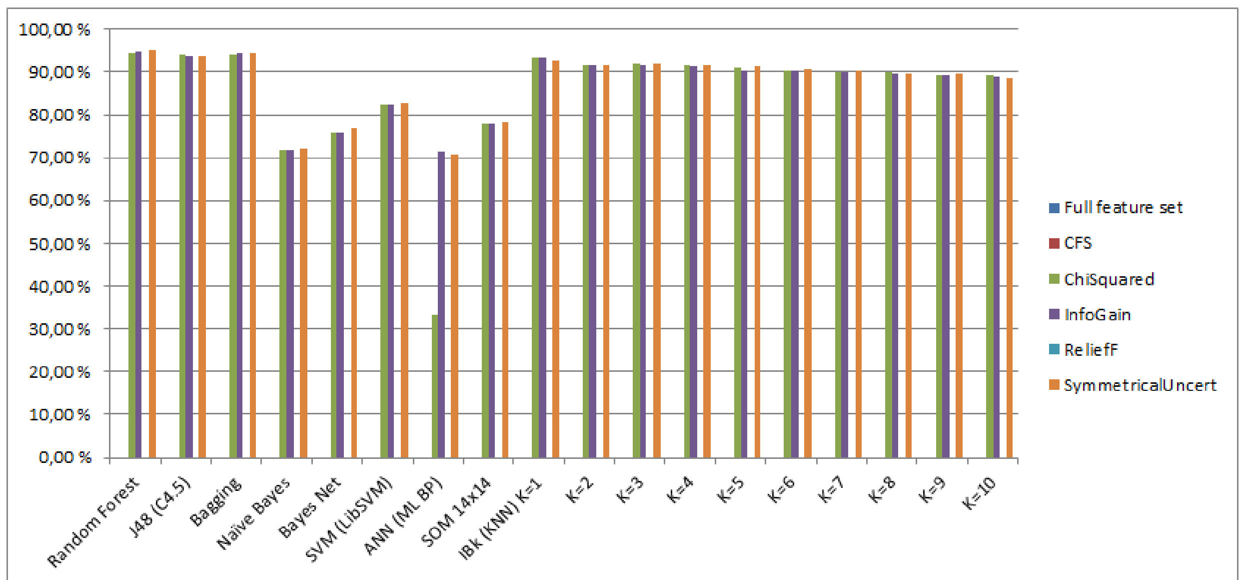

Figure 19: Classifier accuracies for 4-gram

B Datasets

This is examples of the different stages of the data used.

B.1 Asm file example

Asm files contain all extracted assembly code.

```
; PDB File Name : AcroRd32Info.pdb
; OS type   :  MS Windows
; Application type:  Executable 32bit

include uni.inc ; see unicode subdir of ida for info on unicode

.686p
.mmx
.model flat

; ===========================================================================

; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use32
assume cs:_text
;org 401000h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing

; =============== S U B R O U T I N E =======================================

; Attributes: bp-based frame

; int __cdecl sub_401000(wchar_t *Src, wchar_t *Dst, rsize_t SizeInWords)
sub_401000 proc near ; CODE XREF: sub_40164D+30p

Src = dword ptr  8
Dst = dword ptr  0Ch
SizeInWords = dword ptr  10h

push ebp
mov ebp, esp
```

```
push edi
xor edi, edi
cmp [ebp+Src], edi
jnz short loc_401012
xor eax, eax
jmp loc_4010B7
; ---------------------------------------------------------------------------

loc_401012: ; CODE XREF: sub_401000+9
push esi
cmp Memory,edi
jnz short loc_401069
push 208h ; Size
call ds:malloc
pop ecx
mov Memory,eax
cmp eax, edi
jz short loc_401084
mov esi, 104h
push esi ; nSize
push eax ; lpFilename
push edi ; hModule
call ds:GetModuleFileNameW
cmp eax, 3
jbe short loc_401088
cmp eax, esi
jz short loc_401088
push 5Ch ; Ch
push Memory ; Str
call ds:wcsrchr
pop ecx
pop ecx
cmp eax, edi
jz short loc_401088
xor ecx, ecx
mov [eax+2], cx
cmp Memory,edi
jz short loc_401084

loc_401069: ; CODE XREF: sub_401000+19
push 0FFFFFFFFh ; MaxCount
push Memory ; Src
```

```
push [ebp+SizeInWords] ; SizeInWords
push [ebp+Dst] ; Dst
call ds:wcsncpy_s
add esp, 10h
test eax, eax
jz short loc_40109D

loc_401084: ; CODE XREF: sub_401000+2E
; sub_401000+67j ...
xor eax, eax
jmp short loc_4010B6
```

B.2 N-gram example

This is an example of a 4-gram file.

```
push mov push xor
mov push xor cmp
push xor cmp jnz
xor cmp jnz xor
cmp jnz xor jmp
jnz xor jmp push
xor jmp push cmp
jmp push cmp jnz
push cmp jnz push
cmp jnz push call
jnz push call pop
push call pop mov
call pop mov cmp
pop mov cmp jz
mov cmp jz mov
cmp jz mov push
jz mov push push
mov push push push
push push push call
push push call cmp
push call cmp jbe
call cmp jbe cmp
cmp jbe cmp jz
jbe cmp jz push
cmp jz push push
jz push push call
push push call pop
push call pop pop
call pop pop cmp
pop pop cmp jz
pop cmp jz xor
cmp jz xor mov
jz xor mov cmp
xor mov cmp jz
mov cmp jz push
cmp jz push push
jz push push push
push push push push
push push push call
```

```
push push call add
push call add test
call add test jz
add test jz xor
test jz xor jmp
jz xor jmp push
xor jmp push call
jmp push call pop
push call pop mov
call pop mov jmp
pop mov jmp push
```

B.3 Distribution example

An example of the opcode distribution in benign and malicious files.

```
4-gram Malware Benign
['mov', 'mov', 'mov', 'mov'] 1711324 3249554
['mov', 'mov', 'call', 'mov'] 434662 914133
['mov', 'mov', 'mov', 'call'] 434459 1067418
['mov', 'call', 'mov', 'mov'] 373213 790247
['lea', 'mov', 'mov', 'mov'] 232871 177003
['call', 'mov', 'mov', 'mov'] 231107 600093
['mov', 'lea', 'mov', 'mov'] 209694 161836
['push', 'sub', 'mov', 'mov'] 171528 134236
['mov', 'mov', 'lea', 'mov'] 171506 149430
['mov', 'mov', 'mov', 'lea'] 153913 139216
['mov', 'add', 'pop', 'retn'] 141715 38411
['mov', 'mov', 'call', 'test'] 137472 234319
['lea', 'mov', 'mov', 'call'] 137086 74515
['mov', 'mov', 'mov', 'add'] 135989 127476
['sub', 'mov', 'mov', 'mov'] 131318 116573
['mov', 'mov', 'add', 'pop'] 131216 75818
['push', 'push', 'push', 'push'] 123183 315753
['mov', 'call', 'test', 'jz'] 122967 181671
['pop', 'pop', 'pop', 'pop'] 121890 89260
['add', 'pop', 'retn', 'mov'] 118492 39179
['mov', 'jmp', 'mov', 'mov'] 117549 219344
['call', 'mov', 'mov', 'call'] 110186 112394
['cmp', 'jz', 'cmp', 'jz'] 109344 149707
['pop', 'retn', 'mov', 'mov'] 109202 54215
['lea', 'mov', 'call', 'mov'] 107716 76589
['mov', 'call', 'mov', 'call'] 107599 90487
['xor', 'mov', 'mov', 'mov'] 105984 61840
['jmp', 'mov', 'mov', 'mov'] 105721 222397
['test', 'jz', 'mov', 'mov'] 104906 260306
['mov', 'cmp', 'jz', 'mov'] 102761 142307
['mov', 'test', 'jz', 'mov'] 101745 352089
['cmp', 'jz', 'mov', 'mov'] 97918 152609
['mov', 'mov', 'mov', 'cmp'] 97281 140735
['mov', 'mov', 'call', 'lea'] 95882 46472
['mov', 'push', 'sub', 'mov'] 95382 41343
['retn', 'push', 'sub', 'mov'] 94775 81142
['pop', 'retn', 'push', 'sub'] 93440 81539
```

B.4 Arff file example

Example of an arff file used with Weka.

```
@relation '1gram_all-weka.filters.CfsSubsetEval-Sweka.attributeSelection.BestFirst -D 1 -N 5'

@attribute movsxd numeric
@attribute nop numeric
@attribute movnti numeric
@attribute cmovb numeric
@attribute bt numeric
@attribute bts numeric
@attribute cmova numeric
@attribute in numeric
@attribute ror numeric
@attribute repe numeric
@attribute movups numeric
@attribute leave numeric
@attribute stosd numeric
@attribute cld numeric
@attribute prefetcht1 numeric
@attribute movsw numeric
@attribute stosw numeric
@attribute pushf numeric
@attribute fldz numeric
@attribute CLASS {MALWARE,BENIGN}

@data
234,156,24,8,22,25,4,1,1,0,0,0,0,0,0,0,0,0,0,MALWARE
9,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,MALWARE
80,2,0,1,3,7,1,0,1,0,0,0,0,0,0,0,0,0,0,MALWARE
0,8,0,0,0,0,0,0,0,0,3,0,0,0,0,0,0,0,0,MALWARE
38,80,0,2,32,3,0,0,0,0,0,0,0,0,0,0,0,0,0,MALWARE
29,45,24,10,4,3,2,0,0,0,0,0,0,0,0,0,0,0,0,MALWARE
833,1874,24,52,53,74,9,0,1,0,902,0,0,0,0,0,0,0,0,MALWARE
26,265,0,59,107,10,10,0,1,0,0,0,0,0,0,0,0,0,0,MALWARE
95,172,0,8,7,5,0,0,0,0,0,0,0,0,0,0,0,0,0,MALWARE
88,1,0,1,9,7,1,0,1,0,0,0,0,0,0,0,0,0,0,MALWARE
200,0,0,3,5,7,3,0,3,0,0,0,0,0,0,0,0,0,0,MALWARE
39,741,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,MALWARE
88,1,0,1,9,7,1,0,1,0,0,0,0,0,0,0,0,0,0,MALWARE
1444,320,24,19,160,135,17,2,21,0,0,0,0,0,0,0,0,0,0,MALWARE
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,MALWARE
```

```
0,223,0,0,0,0,0,0,0,0,0,245,0,0,0,0,0,0,0,MALWARE
55,0,0,0,4,3,0,0,0,0,0,0,0,0,0,0,0,0,0,MALWARE
12,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,MALWARE
94,154,0,1,34,0,0,0,3,20,0,4,1,0,0,0,0,0,0,MALWARE
652,908,24,11,175,43,8,0,1,4,0,0,1,3,0,0,0,0,0,MALWARE
130,177,0,0,4,4,5,1,0,1,0,1,0,0,0,0,0,0,0,MALWARE
206,128,24,1,34,43,4,0,1,4,0,0,0,0,0,0,0,0,0,MALWARE
88,1,0,1,9,7,1,0,1,0,0,0,0,0,0,0,0,0,0,MALWARE
54,94,0,1,6,3,1,0,1,0,0,0,0,0,1,0,0,0,0,MALWARE
88,1,0,1,9,7,1,0,1,0,0,0,0,0,0,0,0,0,0,MALWARE
3,9,0,0,0,0,0,0,1,0,3,0,0,0,0,0,0,0,0,MALWARE
3,0,0,0,0,0,0,0,0,0,0,1,2,2,0,0,0,0,0,MALWARE
25,108,0,3,80,90,8,8,5,0,0,5,0,1,0,0,0,0,0,MALWARE
50,66,24,1,2,3,3,0,1,0,0,1,0,2,0,0,0,0,0,MALWARE
921,24,0,26,165,74,6,2,20,0,0,0,0,0,0,0,0,0,0,MALWARE
5,62,0,6,4,0,0,0,1,0,0,0,0,0,1,0,0,0,0,MALWARE
30,28,24,1,0,1,3,0,1,0,1,0,0,0,0,0,0,0,0,MALWARE
8,4,8,9,17,4,11,3,1,0,2,0,0,0,0,0,0,0,0,MALWARE
162,128,0,4,11,17,2,0,0,0,2,0,0,0,0,0,0,0,0,MALWARE
```

C Source code

The source code from selected scripts are included in this section. The list is not exhaustive.

C.1 createArff4gram

This script creates the 4-gram files.

```
from os import listdir
from os.path import isfile, join
import time

start = time.time()

# Allowed instructions
allowedInstructions = []

# File names
packedMalware = []
packedBenign = []

cleanMalware = []
cleanBenign = []


# Instructions used by all files in total
allInstructions = []
numberOfInstructionsInTotal = []

numberOfInstructionsMalwareClean = []
numberOfInstructionsMalwareTotal = []

numberOfInstructionsBenignClean = []
numberOfInstructionsBenignTotal = []

# Read which files are packed from file
def read_packed(from_file):
	vars = []
	file = open(from_file, 'r')
	for line in file:
		# Split on ,
		var = line.split(',')
		# Remove path, keep only file name
		vars.append(var[1][8:])
	return vars
```

```
# Read from file
def read_from_file(from_file):
        file = open(from_file , 'r')
        filecontent = [line.strip() for line in file]
        return filecontent

# Read from n-gram file
def read_ngram(path, from_file):
        file = open(path + "\\" + from_file , 'r')
        filecontent = [line.strip() for line in file]
        for ngram in filecontent:
                if ngram not in allInstructions:
                        allInstructions.append(ngram)
                        numberOfInstructionsInTotal.append(1)
                else:
                        numberOfInstructionsInTotal[allInstructions.
                           index(ngram)] += 1

# Read from n-gram file
def read_ngram_2(path, from_file, to_file, type):
        attributes = []
        for x in xrange(0, len(allInstructions)):
                attributes.append(0)
        file = open(path + "\\" + from_file , 'r')
        filecontent = [line.strip() for line in file]
        for ngram in filecontent:
                attributes[allInstructions.index(ngram)] += 1
        file2 = open(to_file, 'a')
        file2.write(type)
        for x in xrange(0, len(allInstructions)):
                file2.write("," + str(attributes[x]))
        file2.write("\n")
        file2.close()

        if from_file[:-4] not in packedMalware:
                from_file2 = from_file[:-3]
                from_file2 += "exe"
                if from_file2 not in packedBenign:
                        file3 = open("4gram_not_packed.arff", 'a')
                        file3.write(type)
                        for x in xrange(0, len(allInstructions)):
                                file3.write("," + str(attributes[x]))
                        file3.write("\n")
                        file3.close()

# Read allowed instructions
allowedInstructions = read_from_file("allowed_instructions.txt")

# Read which malware files are packed
```

```
packedMalware = read_packed("packed_malware.csv")
# Read which benign files are packed
packedBenign = read_packed("packed_benign.csv")

# Read malware files
malwarePath = "..\\ngrams\\malware\\4-gram"
malwareFiles = [ f for f in listdir(malwarePath) if isfile(join(
    malwarePath,f)) ]
for malwareFile in malwareFiles:
        read_ngram(malwarePath, malwareFile)

# Read benign files
benignPath = "..\\ngrams\\benign\\4-gram"
benignFiles = [ f for f in listdir(benignPath) if isfile(join(
    benignPath,f)) ]
for benignFile in benignFiles:
        read_ngram(benignPath, benignFile)

# Print to file
file2 = open("4gram_all.arff", 'w')
file2.write("@RELATION 4gram_all" + "\n\n")
file2.write("@ATTRIBUTE CLASS {MALWARE,BENIGN}" + "\n")
for x in xrange(0, len(allInstructions)):
        file2.write("@ATTRIBUTE " + str(allInstructions[x].replace("\
            t", "-")) + " NUMERIC" + "\n")
file2.write("\n" + "@DATA" + "\n")
file2.close()

# Print to file
file3 = open("4gram_not_packed.arff", 'w')
file3.write("@RELATION 4gram_not_packed" + "\n\n")
file3.write("@ATTRIBUTE CLASS {MALWARE,BENIGN}" + "\n")
for x in xrange(0, len(allInstructions)):
        file3.write("@ATTRIBUTE " + str(allInstructions[x].replace("\
            t", "-")) + " NUMERIC" + "\n")
file3.write("\n" + "@DATA" + "\n")
file3.close()

# Get the full attribute set
for malwareFile in malwareFiles:
        read_ngram_2(malwarePath, malwareFile, "4gram_all.arff", "
            MALWARE")
for benignFile in benignFiles:
        read_ngram_2(benignPath, benignFile, "4gram_all.arff", "
            BENIGN")

end = time.time()
print end - start
```

C.2 copyAllExe

This script creates a bat file that finds and copies all exe files.

```
# This script creates a bat file that copies all exe files on the C:\
    drive to C:\exe
import os
# The bat file we are creating
file = open("C:/Users/Simen/Documents/copyAllExe.bat", "w")
n=1
# Recursively going through all files and folders
for (dir, _, files) in os.walk("C:\\"):
        for f in files:
                if len(f) > 3:
                        # If exe file
                        if f[-4:] == '.exe':
                                path = os.path.join(dir, f)
                                # Print command for copying to
                                    destination
                                idaPath = "COPY␣" + path + "␣C:\\exe"
                                n += 1
                                file.write(idaPath)
                                file.write("\n")
print n
```

C.3 createBatAllBenign

This script creates a bat file that contains the IDA commands for creating asm files for all the files in a given folder.

```
# This script creates a bat file that contains the IDA commands for
   creating asm files for all the files in a given folder
# Is used for the benign samples
from os import listdir
from os.path import isfile, join
from os import popen
import subprocess

mypath = "C:\\exe"
print mypath
# Go through all the files in the folder
onlyfiles = [ f for f in listdir(mypath) if isfile(join(mypath,f)) ]

print onlyfiles
# The bat file we are creating
file = open("C:/Users/Simen/Documents/allBenign.bat", "w")
n=1
for everyfile in onlyfiles:
        # Add the IDA command and file
        path = "idaw64␣-B␣C:\\exe\\" + everyfile
        print path
```

```
        file.write(path)
        file.write("\n")
        if n%10 == 0:
                # Add echo for every tenth to see the progress when
                    running the bat
                file.write("echo␣" + str(n) + "\n")
        n += 1
file.close
```

C.4 getInstructionsFromAsm

This script reads a list of allowed instructions from file. It then prints all instructions from all asm files in a given folder to a file.

```
# This script reads a list of allowed instructions from file.
# It then prints all instructions from all asm files in a given
    folder to a file.
from os import listdir
from os.path import isfile, join

allowed_instructions = []

def read_instructions_file(from_file):
        file = open(from_file , 'r')
        filecontent = [line.strip() for line in file]

        for line in filecontent:
                if line not in allowed_instructions:
                        allowed_instructions.append(line.lower())

read_instructions_file("instructions-a.txt")
print allowed_instructions

instructions = []

def read_from_file(from_file):
        read_list = []
        file = open(from_file , 'r')
        filecontent = [line.strip() for line in file]

        for line in filecontent:
                line_args = line.split()
                read_list.append(line_args)

        return read_list

mypath = "C:\\test\\asm"
onlyfiles = [ f for f in listdir(mypath) if isfile(join(mypath,f)) ]

for everyfile in onlyfiles:
```

```
        path = "C:\\test\\asm\\" + everyfile
        asm_file = read_from_file(path)

        for x in asm_file:
                if len(x) > 0:
                        if len(x[0]) > 1:
                                if x[0].lower() in
                                    allowed_instructions:
                                        if x[0].lower() not in
                                            instructions:
                                                print x[0]
                                                instructions.append(x
                                                    [0].lower())

        print len(instructions)

file2 = open("allMalwareInstructions.txt", "w")
for inst in instructions:
        file2.write(inst)
        file2.write("\n")
file2.close

print "done"
```