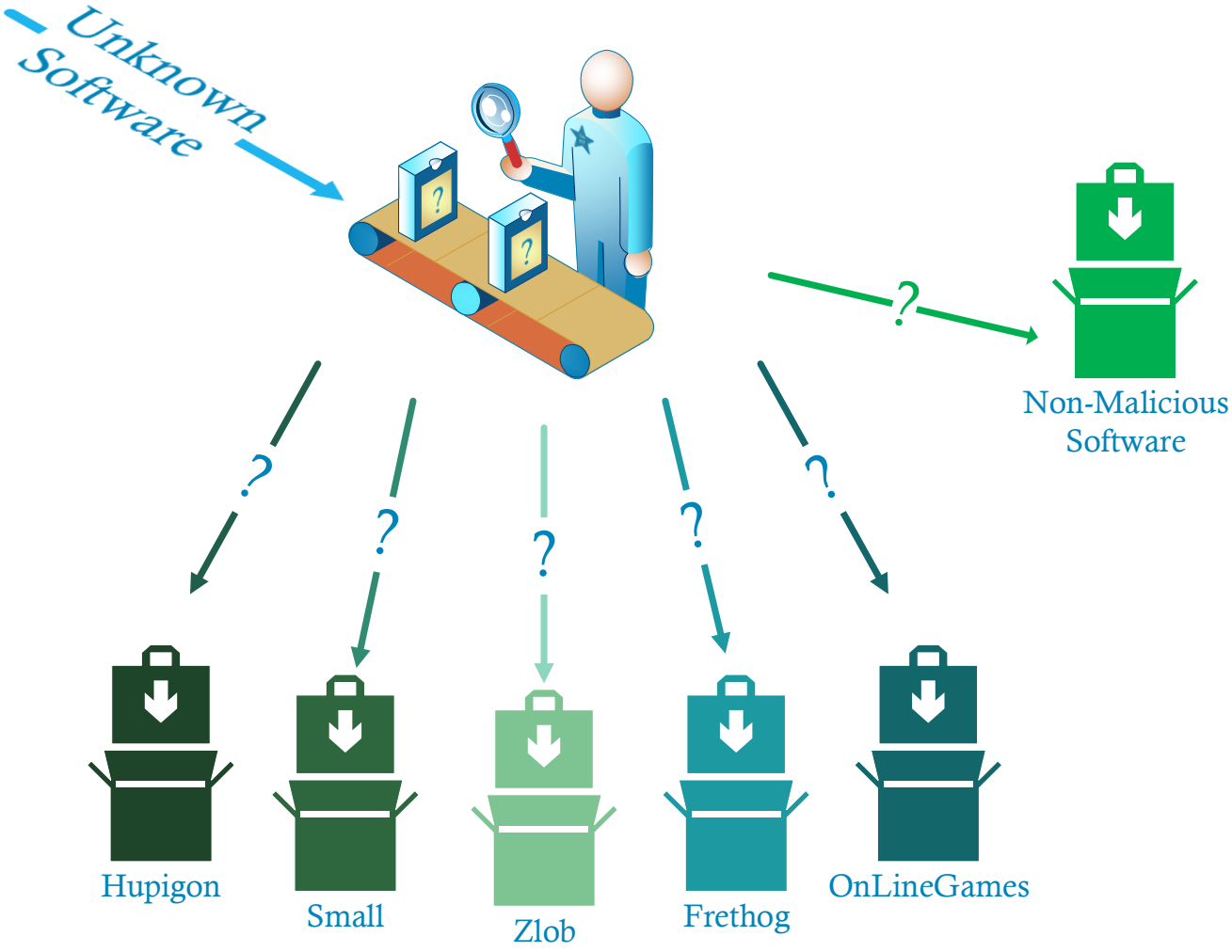
MASTER THESIS

Dynamic Malware Analysis: Detection and Family Classification using Machine Learning



Aalborg University
P10, Spring Semester 2015
Group 1023 - Networks & Distributed Systems

AALBORG UNIVERSITET

Institute of Electronic Systems
Department of Communication Technology
Niels Jernes Vej 12 A5
9220 Aalborg Øst
Phone 9635 8650
http://es.aau.dk

Title:

Dynamic Malware Analysis: Detection and Family Classification using Machine Learning

Theme:

Master Thesis
Networks & Distributed Systems

Projectperiod:

P10, spring semester 2015

Group:

1023

Members:

Steven Strandlund Hansen

Thor Mark Tampus Larsen

Supervisors:

Jens Myrup Pedersen
Matija Stevanovic

Number of pages: 153

Number of appendix pages: 19

Number of annex pages: 1 (CD)

Finished: 3rd of June 2015

Abstract:

This study is a continuation of a earlier research project made by members of this group, which aims to solve the problem of the increasing growth of malware each day. This research performs both detection and family classification based on dynamic analysis using machine learning. By improving and utilizing the analysis setup from the previous research, a customized version of Cuckoo, analyzes approximately 200,000 malware, using a total amount of 30 VMs. To cope with the large sample set it has been implemented in a scalable and distributed manner. In addition, a smaller setup was made to analyze approximately 850 cleanware samples, needed for detection.
API calls with its specified input arguments was used as features to represent a combination matrix including the following representation techniques: sequence, frequency and binary. Random Forests, are injected with features and labels, based on behavioral information extracted from malware, detected by Microsoft. Feature selection is performed based on Information Gain Ratio, to remove redundant and irrelevant features.
The detection gave a weighted average TPR, PPV and AUC of 0.969, 0.970 and 0.996, respectively. In addition, the family classification gave weighted average TPR, PPV and AUC of 0.865, 0.872 and 0.977, respectively. From the results, it was concluded, that detection and family classification can indeed be done based on dynamic analysis using Random Forests. It is believed, that this study can help solve the issue of dealing with the great amount of new malware, that emerge every day.

PREFACE

This report has been written by group 1023, who are 4th semester students of the Networks & Distributed Systems masters program under the Department of Electronic Systems at Aalborg University. This Master Thesis is a continuation of a 9th semester project carried out by members of this group. The subject of this project is "Dynamic Malware Analysis: Detection and Family Classification using Machine Learning". The project has been carried out in the period: February, 1. to June, 3. 2015.

Reading Instructions

The report created during the project period is addressed to supervisors and other students. This report presumed the reader to have a basic knowledge within the field of networking and programming.

This report includes 7 Parts. It is built up by the following: Problem Analysis, Technical Analysis, Design & Implementation, Results, Conclusion, References and Appendices. Each major chapter will include a Chapter Summary with the purpose of providing the reader with a overview of the contents and the take-away messages. Furthermore an article explaining the previous project has been attached to the end of the report. Included in this report is a CD which contains all the material, that has been used in this project. The report contains references in the Harvard-method format which includes; [Surname, Year]. The reference points to a bibliography in Chapter 13 and will include information about the source, as author, year of release and appropriate URL. The figures, tables and listings are numbered according to their location in the report, i.e. the chapter. If the reference is not on the same page as what it refers to, it will include page numbering to where it is placed.

Abbreviations used throughout the report is included right after the table of contents.

INTENTIONALLY LEFT BLANK

CONTENTS

INTENTIONALLY LEFT BLANK

ABBREVIATIONS

API: Application Programming Interface

ARFF: Attribute-Relation File Format

AUC: Area Under the Curve

AV: Anti-Virus

BSON: Binary JavaScript Object Notation

CARO: Computer Anti-virus Research Organization

CDCBF: Class Driven Correlation based Feature Selection

CFSS: Correlation-based Feature Subset Selection

CS: Chi-Square

DB: DataBase

DFSF: Unbalanced Data Feature Selection

DLL: Dynamic-Link Library

DNS: Domain Name Service

ENISA: European Union Agency for Network and Information Security

F-M: F-Measure

FCBF: Fast Correlation-Based Filter

FN: False Negative

FP: False Positive

FPR: False Positive Rate

FS: Feature Selection

IE: Information Entropy

IG: Information Gain

IGR: Information Gain Ratio

JSON: JavaScript Object Notation

ML: Machine Learning

MSE: Microsoft Security Essentials

NCSI: Network Connectivity Status Indicator

NIC: Network Interface Controller

NN: Neural Networks

PE: Portable Executable

PPV: Positive Predictive Value

PUP: Potentially Unwanted Software

R/W: Read/Write

RAT: Remote Access Tool

REG: Registry

RegEx: Regular Expression

RF: Random Forests

ROC: Receiver Operating Characteristic

SVM: Support Vector Machine

TLS: Transport Layer Security

TN: True Negative

TP: True Positive

TPR: True Positvie Rate

UAC: User Account Control

VM: Virtual Machine

WEKA: Waitkato Environment for Knowledge Analysis

XOR: Exclusive OR

INTENTIONALLY LEFT BLANK

Part I

Problem Analysis

INTENTIONALLY LEFT BLANK

CHAPTER 1

INTRODUCTION

Surfing the Internet, using social media and sharing information back and forth, is not as safe as it used to be. In line with the exponential growth in communicated data we, as the users, are more dependent on the Internet than ever. With the increased Internet usage, states, companies and private users have to secure themselves even better, in order to protect vulnerable information. During the last couple of years, more attention has also been put on Cyber Security. On May 21, this year (2015), European Union Agency for Network and Information Security (ENISA) held a conference, where some of the goals were to ensure that more focus are put on the nature of Cyber Security involving technology, business and policy. Additionally they emphasized that there is a need for a constantly educated and updated workforce, government and businesses, see [ENISA, 2015]. It emphasizes that there is an increased need of security, especially when it comes to malicious software and when major vulnerabilities are found. One of the latest and largest vulnerabilities found, was the *Heartbleed-bug*, where cyber criminals exploited a serious bug in the OpenSSL data encryption. The bug was found in the Transport Layer Security (TLS) heartbeat function. Here perpetrators could access up to 64 kilobytes of the application memory, at every heartbeat, i.e. every time a client checks if the server is still operational, thus the name Heartbleed, see [Codenomicon, 2014]. Potential user names and passwords, emails, and business critical documents could be stored in the memory. New malicious programs, or malware, are registered every day, with one of the latest major types being the ransomware *CryptoWall*. This type of malware lock and encrypt programs and files on the infected system, where people are scammed into paying for the system to be unlocked. This typically includes messages stating that FBI or other federal agencies have detected violations from the computer according to federal laws, see [FBI, 2015].

There will always be threats and vulnerabilities, which malware authors will exploit. Therefore, it is important for security companies to detect the malicious programs and notify companies and users about potential vulnerabilities. In line with the exponential growth of the Internet, the number of new malware is increasing everyday, which has become difficult to analyze manually. A bar plot is depicted in Figure 1.1 (p. 4), which shows the trend of the growth of malware. Here it can be seen that both the total amount of malware and new malware have an exponential growth. The growth is not constant but by studying the graph, it can be concluded that the malware more than doubles every second year and actually triples from 2012 to 2014.

Analyzing the increasing number of malware, requires a lot of human resources if done manually. As of 2015, the AV-Test institute registers 390,000 new malicious programs every day, which is infeasible to analyze manually, see [AV-test, 2015]. Even more, the malware have to be divided into groups or families to which their code and behavior corresponds to. Therefore, more recent methods of analyzing malware apply, what is called dynamic analysis, in comparison with the traditional method called static analysis. Manual static analysis of malware requires human interaction at each sample and more so, many hours of

experience in reverse engineering, decompilation, disassembling and decryption. Dynamic analysis relies on recording the behavior of malware after being executed in a virtual environment, and using machine learning one can predict the nature of the malware.

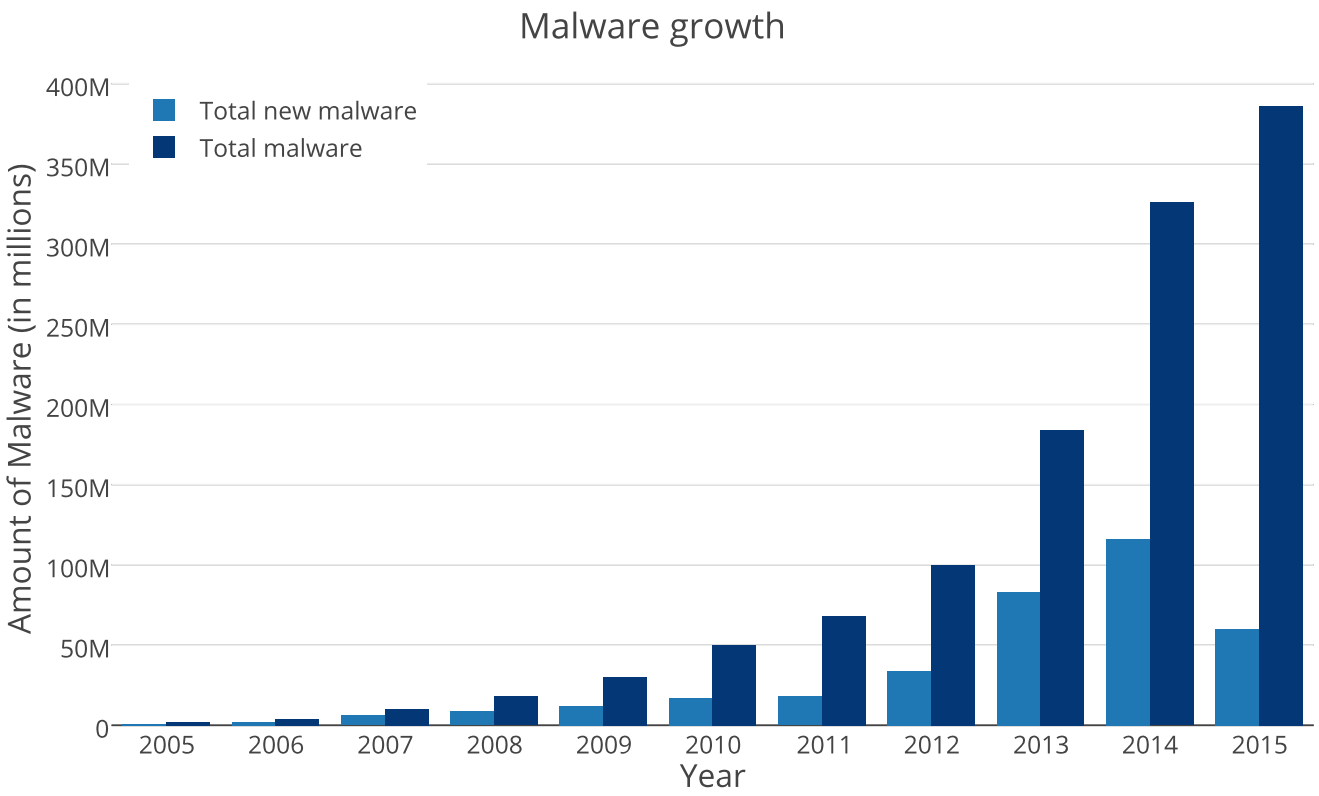


Figure 1.1: New and total amount of Malware on a per year basis [AV-test, 2015].

As malware become more and more sophisticated, various obfuscation and encryption techniques are used, which makes it even harder for static analysts to detect and clarify their behavior. For instance, commonly used methods for obfuscation includes logical operations like Exclusive OR (XOR) or simple letter substitution, which easily hides content from untrained eyes. Additionally, some malware authors use Runtime packers, where the entire malware program is obfuscated until it is placed in the memory. Packers compress data, which makes the data unreadable before it has been decompressed. If the malware authors have created their own compression algorithm, the code is very difficult to read without a wrapping program, that will decompress the program in memory and thus reveal the original code, see [Malwarebytes, 2013]. These problems are solved using dynamic analysis, as long as the malware does not detect that it is being monitored.

This study addresses the presented challenges with the goal of utilizing machine learning as a data mining and prediction tool, for more efficient, and potentially more precise detection and classification of malware.

CHAPTER 2

PREVIOUS RESEARCH

This chapter will include a summary of the previous work by this group, see [Larsen et al., 2014]. The summary can be followed in Figure 2.1. After the summary, this project will position itself in relation to the last project. Furthermore, an article about the previous project has been attached to the end of the report, if the reader is interested in a more thorough description.

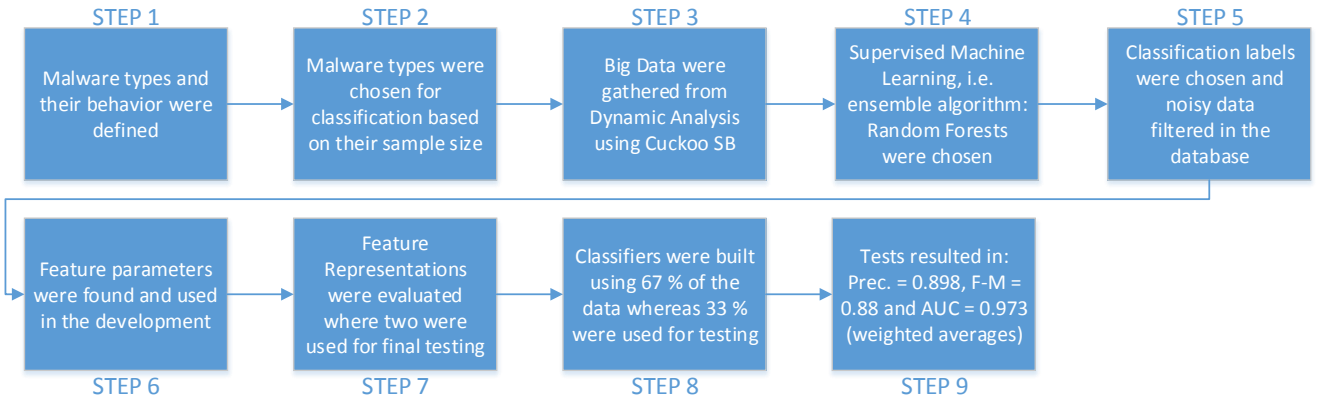


Figure 2.1: Time-line of the progress of the last project.

In the previous study [Larsen et al., 2014], the starting point for understanding malware behavior, was to define the different malware types: Trojan Horse, Worm, Virus etc. Based on the definitions, it was concluded that the types could be used to differentiate different behaviors, which is the primary goal in classification. Then, an analysis was made on data-collection to clarify what kind of information to use for the classification, namely: Data based on static or dynamic analysis. The decision was made on dynamic analysis, meaning that the malware behavior was recorded in run-time after being executed in a sandbox environment. The benefits of a dynamic approach in contrast to static was discussed in Chapter 1. Cuckoo Sandbox was chosen, which is open-source and easy to use. After adjusting and modifying the system to fit the needs of the project, a classification method had to be found.
As malware **type** classification was chosen, the supervised learning algorithm Random Forests (RF) was applied on the data output from approximately 42,000 malware samples. Before applying supervised machine learning, labels for the malware samples had to be extracted, and noise removed from the data. Labels were extracted from Cuckoo, since the malware samples are scanned by several Anti-Virus (AV) programs by uploading it to VirusTotal. Here, problems were experienced as the labels from the AV programs were not always consistent. Furthermore, a *whitelist* was generated to sort out the data to be used in the classification. This was necessary since the Big Data output from Cuckoo contained non-malicious recordings generated from programs installed in the Virtual Machine (VM), e.g. Skype, Flash, Adobe, etc. The features used in the last study were: Accessed files/REG keys, mutexes, Domain Name Service (DNS) requests and API calls.
Using RF, several feature representations were tested. The features were the parameters mentioned before,

whereas different data or representations were used for each of them. This included: Binary - true/false if the API was present, Frequency - number of times the API occurs, Bins - a feature construction of the API frequency, Sequence for the API calls and several counters constructed from the remaining parameters. RF with 160 trees were applied using 4 classes, namely: Trojan, Potentially Unwanted Software (PUP), Adware and Rootkit. The results were satisfactory, showing a weighted average precision, F-Measure and Area Under the Curve (AUC) of 0.898, 0.88 and 0.973 respectively. The previous study can be found on the CD in the folder `...\Previous Study` and the paper can be found attached to the end of this report.

2.1 Goals in this Project

One of the main problems met in the previous research, were the potentially ambiguous labels. Here, some malware types could execute overlapping behaviors, leaving them hard to discriminate. This would lead to miss-classifications and lower prediction rate. This study will try to address this problem by utilizing other labels.
Another problem was that, the classification assumed that all samples were malicious. This meant that if unknown software were injected, it would be classified as malware no matter what. This study will try to perform malware detection based on dynamic analysis. Other problems that needs to be addressed, is the amount of used data. Building models based on larger data sets will ensure a more robust classification. Additionally, the data should be as uniformly distributed as possible, leading to a more fair classification. At last, this study will try to improve the classification rate by utilizing other features and actually using a feature selection algorithm, which potentially can remove irrelevant and redundant features. The aforementioned aspects will be the main focus in this project. To summarize the main differences, the following list has been made. The goal is to extend and improve the system by:

1. Performing malware detection.
2. Performing classification of malware families.
3. Finding new and improving old features.
4. Applying a feature selection algorithm, that will select the most discriminative features.
5. Building a large database of malware by collecting more samples.
6. Retrieving a uniform sample set among the malware classes.

The goal of this chapter was to give the reader an idea of the previous study, and to address the problems in the aforementioned. This included a summary of the progression in the last project along with the improvements this project aims to address. The next chapter includes the Problem Isolation.

CHAPTER 3
PROBLEM ISOLATION

This chapter will include the preliminary research, which aims to elucidate the goals of this project. The goals and the differences in relation to the last project was summarized in Section 2.1. The preliminary analysis presented in this chapter will narrow down the problem at a top level technical detail, which will lead to the technical problem description and statement in Chapter 4. This includes research in Malware Detection, Families and Types, AV-program, the CARO naming standard and Malware Database.

3.1 Malware Detection

This section will first give a brief clarification of the term cleanware in comparison to malware along with a reason of why detection are of great importance. Furthermore, this section presents an overview of the field, with the goal of deciding, which approach to use in this project. Thus this will sole the first goal of the project:

- *Performing malware detection.*

Detection can involve both Behavior-based and Signature-based detection. The main differences of these two approaches will be discussed and a technique decided upon.

3.1.1 Cleanware and Malware

The term cleanware is used when talking about benign software, which indicates non-malicious activity of the file or program. In many cases it is useful to discriminate malware and cleanware, in order to ensure an unknown file, is not malicious. Examples could be:

- Downloading a file.
- Opening attached components in an E-mail.
- Inserting a memory dongle to your system.
- Background detection on your computer.

This project aims to perform malware detection, where the trained models can discriminate cleanware from malware. Being able to do this without specific signatures from the malware source-code, which are normally used by AV programs, the system could potentially detect zero-day malware. Here the term **zero-day** are used for new malware not yet detected or registered.

3.1.2 Behavior-based Detection

As the name suggests, Behavior-based malware detection relies on the analysis of the malware behavior during its run-time, see [Wu et al., 2011]. Analyzing malware after its execution, and in a confined environment, is typically referred to as dynamic analysis, see [Larsen et al., 2014] for further details about

what tools have been researched in the last study. The advantage when using this type of detection is to be independent of the actual source-code of the malware. Here the source-code can execute different code obfuscation techniques e.g. packing their code, polymorphism and metamorphism. These are briefly explained in Subsection 3.1.3 and will therefore not be discussed further in this section.

3.1.3 Signature-based Detection

Signature-based malware detection, commonly refers to static analysis, where the malware sample is analyzed and unique signatures extracted. These can then be used to discriminate malware files from benign files and is a commonly used method by AV-vendors, see [Wu et al., 2011]. The problem arises, when dealing with code obfuscation techniques used by the malware. Some of the obfuscation techniques used against Signature-based Detection are listed below:

- **Packing:** Here, the malware is compressed into a binary file, which are only readable if correct decompression is used. When decompressed, it is loaded into memory in a human readable form. Malware can be compressed in many ways and even several times making it close to impossible to reverse-engineer the code, see [Malwarebytes, 2013].
- **Polymorphism:** Here, the malware changes part of its code after every iteration it runs on the computer, while another part remains the same, see [Sikorski and Honig, 2012].
- **Metamorphism:** Here, the malware rewrites all its code after each iteration it runs on the computer, but still keeps the same functionality, see [Sikorski and Honig, 2012].

Reviewing the problem of malware utilizing obfuscation techniques, **this project will utilize Behavior-based malware detection along with classification**. The next section will describe malware families and malware types, where last mentioned was the focus in the last study.

3.2 Malware Classification

This section will explain the ideas behind family classification and type classification. In the past research performed by members of this group, type classification were used in Random Forests. As mentioned earlier this study will deal with family classification. Therefore, this section deals with the goal:

- *Performing classification of malware families.*

This will be done with the hope of achieving a better predictive performance. Furthermore, reasonable assumptions will be made about the classes, since it is impossible to predict exactly how the malware samples will behave in a large scale dynamic analysis. In the following, malware types and families are explained.

3.2.1 Types

A malware type is a group of malware, whereas the name typically suggests how it infects, propagates or exploits the compromised machine. For instance, the name Trojan Horse suggests that the malware has infected the machine by masking its malicious intent, as a useful and non-malicious program. In the same

way, the name Spyware or Adware suggests that user information is logged with the purpose of selling the information, or to show advertisements based on what fits to the particular user. This intuitively makes sense, as the type suggests a certain behavior. Types were used for the classification in the last study, see [Larsen et al., 2014], where an explanation of the specific types can be found. The problem arises when security companies, label malware which execute behaviors that overlap other types. The consequence of this, when performing classification, is an ambiguous classifier. To elaborate more on this, an example can be seen in Figure 3.1.

Figure 3.1: Example of a type Trojan that also executes Adware behaviors, [Wiki-security, 2015].

This is a major problem when classifying malware using the types named by AV-programs, since these cases will output ambiguous behaviors. Conclusively, it will give rise to many false positives/negatives in the classification. In other words, it becomes very challenging to distinguish the classes as the models for each class are built from samples, that could also include a second, third or even fourth type. It can be discussed that providing better features or even more samples for each class could help solve this issue. Though, it will still cause a problem if some malware consist of several types, since the classification denotes one unique type for each sample.

3.2.2 Families

A malware family is a group of malware that typically originates from a single source code, and are typically branched into new versions or updates with slightly different behaviors. Even though e.g. two samples from the same family potentially have slightly different behaviors, it is in this study assumed that the similarity of their individual behavior is great enough for classification. The following conclusions from the last study are emphasized and was also discussed in Subsection 3.2.1:

- It was observed from a confusion matrix, that some of the **types must have had similar behavior** because of the large amount of false positives/negatives.
- It was suggested that several samples **could execute the behavior of more than one type**, e.g. the type Trojan could also include the behavior of Adware. This is also suggested by [Wiki-security, 2015].

As this had an impact on the individual class predictions, it is a motivation to use family classification, which potentially could resolve this issue. This was also discussed as one of the goals of this study in Section 2.1.

The downside, in relation to an end-product and in comparison with types, is the great amount of families that exist. At this moment there is a number of 236 major malware families registered by Microsoft, whereas the complete list includes many thousand families, see [Microsoft, 2014]. The consequence of

many classes in relation to supervised learning, is that more discriminative features and samples are needed to ensure high classification performance. This study will not take all families into account, but as proof-of-concept, only consider the families where enough samples are present and as uniform as possible.

For type classification, a larger sample space might reduce the number of incorrectly classified instances, but it is believed that it cannot completely eliminate the problem, since malware can include several types. For this reason, **this study will focus on malware families**, as the extracted behavior, are assumed to have higher discriminative power. The reasoning is, that each of the family samples should correlate better with other samples of the same family. This assumption is based on the fact that most AV-programs, label malware by static code analysis, which indicate that they either come from the same source code or have a very similar behavior.

The next section will include a discussion about the issues when extracting the labels and argue which AV-program will be used in this project.

3.3 Naming Malware

One of the difficult tasks experienced in the last study, was extracting labels from the AV-programs. Here, the choice of AV program, was solely based on the amount of types detected along with how many samples were included for type. The extraction of the labels were complicated, since each AV-vendor use different standards for naming the malware together with a potentially inconsistent syntax. To prevent these issues, the labels are in this study extracted from an AV program, that follows a well described naming standard.

The program chosen for this study is Microsoft Security Essentials (MSE), which uses the well known Computer Anti-virus Research Organization (CARO) naming scheme. The following section describes the details surrounding the naming scheme, together with their definition.

3.3.1 CARO Naming Standard

CARO is an informal malware naming scheme developed by individuals from AV companies and researchers, see [CARO, 2014]. Note here, that in relation to the discussion of ambiguous class labels, see Subsection 3.2.1, the naming convention does **not** resolve the problem, but instead tries to resolve the inconsistent labeling.

The idea is to create a common standard, or syntax, for naming malware, to prevent confusion of definitions among e.g. AV-vendors and users. The most complex form is as following:

```
<malware_type>://<platform>/<family_name>.<group_ name>.<infective_length>.<sub-variant><devolution><modifiers>
```

All the conventions here are optional except the family name, since not all entries necessarily are available.

This convention is used by Microsoft and in their AV software namely, MSE or the Win8 version, Windows Defender. For MSE, the scheme used is as following:

```
<malware_type>://<platform>/<family_name>.<sub-variant>!<vendor-specific_comment>
```

It is noticed here, that `infective_length`, `group_name` and `devolution` is not applied in their convention. Additionally, the `modifiers` have been replaced with `!vendor-specific_comment`, which is part of the `modifiers` parameter also used in CARO. To give an idea of the structure a real example is listed below:

```
Backdoor:Win32/Hupigon.D
```

Note here that the vendor-specific comment is optional, which is why it will not be available for all malware. In the following, each term in the above naming convention, will be explained in relation to MSE, [Microsoft, 2015d].

Malware type:
Denotes the appropriate type name for the malware, like Trojan, Worm, Adware etc.

Platform:
Denotes which platform the malware is compatible with, like win32, but can also denote the programming language or file format.

Family name:
Denotes which family the malware type belongs to. This name varies according to the AV-vendors.

Sub-variant:
Describes the version or member of a family denoted with letters in alphabetic order. Usually the order denotes when it was discovered in relation to other variants of the same family, i.e. ".A" came before ".B" and so on.

Vendor-specific comment:
This field is usually used to describe specific files or components, used by other threats in relation to this threat. This might be different for other AV-programs.

This concluded the section describing how to name malware. The next section will present the Malware Database.

3.4 Malware Database

This section will deal with the last two goals of the project, which were listed in Section 2.1, namely:

- *Building a large database of malware by collecting more samples.*
- *Retrieving a uniform sample set among the malware classes.*

3.4.1 Expanding the Database

Because this study includes detection, benign software, known as cleanware are needed besides malware. The next two paragraphs explains, where these two sample sets were collected.

Malware Samples

As mentioned earlier, this project aims to build a larger database of malware samples, so potentially more robust classification models can be created. Along with the approximately 80,000 malware samples (captured in 2014) retrieved from [VirusShare, 2014] in the last project, an additional 190,000 samples are analyzed in this project. The 190k samples were downloaded as a snapshot from a Ukrainian website before it was taken down, where the newest samples are from 2010, see [VX Heavens, 2010]. Because this is five years ago, the majority of the samples are detected when Cuckoo uploads it to VirusTotal, see [VirusTotal, 2014]. This will ensure that more labels can be used in this study. It is known that these samples are outdated, but they can still be used together with the new malware from 2014, such that the whole sample set will exhibit more diverse behavior. This can be reasoned, since some of the same malware families can be found in both sample sets, meaning that the samples will exhibit the same behaviors but with changes in code and structure. The total amount of malware samples in the DataBase (DB) are approximately 270k. Note here that when it comes to choosing samples based on appropriate labels, this number will be lower.

Cleanware Samples

Since this study covers detection based on behavior, cleanware also needs to be collected. It was quite difficult to gather a large enough sample set, because no large DB of free small-sized cleanware programs, could be found. Because a lot of research is done on malware, it was easier to find websites, that would share the samples. It should also be clarified that much of the malware that exist are different versions of the same source code. Malware usually does this in order to avoid being detected by its hash or signatures. This means, that it can be assumed that some of the malware in the DB are the same, but just slightly updated. The amount of cleanware samples are therefore much lower than the amount of malware. The cleanware samples used in this study was retrieved from three individual sources, namely: [Ninite, 2015], [Lupo PenSuite Collections, 2015] and native programs in Windows 7. The amount of samples from each source, are 80 for Ninite, 657 for LupoPenSuite and 104 from the Native Windows programs. This gives a total of 841 samples, which is a small amount compared to the sample set of malware. Though, each of the cleanware programs in the sample set are completely different in terms of the program code and might therefore be more discriminative. Of course, some of the samples include some of the same behavior, since they are within the same category. This could e.g. be web browsers, however each of the web browsers originates from different source codes. Compared to malware, each sample from the same family is assumed to originate from the same source code, and the majority of the behavior from these samples will therefore be more similar.

3.4.2 Uniform Sample Set

Having a sample set distributed in a uniformly manner among the classes, will make the classification more fair. If one class dominates, it is assumed that the results will be skewed. In other words, it is possible to weight the results for each class evenly, and the results are thereby more reliable and can be evaluated fairly. On the other hand, this study also aims to use all the samples available for the classes, and thus slight imbalance will be present. Note here, that the difference between min. and max sample

size will be much lower than the previous study, thus more uniform.

For detection, an equal amount of cleanware and malware is the goal, but this was difficult and time consuming to achieve. Alternatively, by only taking part of the full malware sample set the goal is fulfilled. Because some malware originates from the same version, a larger amount of malware will be used in comparison to cleanware. By same version, it means that it is generally the same malware, but different variants as also explained previously. Regarding the malware classification, the uniformity in samples will be achieved by taking a number of families which have approximately the same number of samples.

3.5 Chapter Summary

This chapter presented an overview of the subjects that addressed the goals presented in Section 2.1. The conclusions in this chapter will narrow the project focus, which is further described in the next chapter, see Chapter 4. Below a summary of the conclusions can be found in point form:

1. **Malware detection:**
   — Use of behavior-based detection by discriminating cleanware and malware.
2. **Malware classification:**
   — Use of family classification to avoid ambiguous classes.
3. **Naming Malware:**
   — Use of MSE AV-program since it uses the CARO naming standard.
4. **Malware Database:**
   — Use of approximately 270k malware and 841 cleanware samples.
   — Use of samples distributed in a uniformly manner.

The next chapter will present the Technical Problem Description, that based on the Problem Analysis, needs to be solved in order to fulfill the goals in Section 2.1.

INTENTIONALLY LEFT BLANK

CHAPTER 4

TECHNICAL PROBLEM DESCRIPTION

Based on the conclusions of the Problem Analysis, this chapter will summarize and specify the technical problems, which has to be solved according to the goals in Section 2.1. The upcoming paragraph motivates the project, which will lead to the overall problem statement. The problem will be divided into several sub-problems, which will be solved in Part II.

Because the growth of malware increases every day, manually analyzing the new malware is very time consuming. Analysis is though, still necessary for coming to grips with the cyber criminals. This study aims to find a more efficient way of detecting and classifying malware, and still provide accurate results. Since the objectives or goals for this study were made based on the results from the previous study, the Problem Isolation (Chapter 3) has introduced the aspect of extending the system to perform detection followed by analysis of the overall issues. By overall issues it means that no low-level technical solutions have been analyzed but instead the top-level problems. This included discussions of: Definitions of malware detection, malware classification together with the use of a CARO compatible AV-program and expanding the collection of malware samples in the DB along with a new cleanware collection. The following highlighted goals were involved:

1. **Performing malware detection.**
2. **Performing classification of families.**
3. Finding new and improving old features.
4. Applying a feature selection algorithm, that will select the most discriminating features.
5. **Building a large database of malware by collecting more samples.**
6. **Retrieving a uniform sample set among the malware classes.**

Note here, that technical methods and solutions of the goals will be presented in the Technical Analysis, see Part II.

The next section will include a technical problem statement which will emphasize the main problem. Furthermore, sub-problems will be defined to specify what to be solved to achieve the final solution.

4.1 Problem Statement

The following paragraph and subsection includes the problem statement and the underlying problems formed on the basis of the Goals in Section 2.1 and the Problem Isolation in Chapter 3.

How to perform Malware Detection and Family Classification using Machine Learning?

4.1.1 Sub-Problems

This section includes the sub-problems considered in this project in relation to the overall problem.

The following technical sub-problems have been formulated:

1. How to perform malware detection (goal 1)?
2. Which malware families should be used as class-labels (goal 2, 5 and 6)?
3. Can new features be created using DLL, Mutexes and I/O arguments for API (goal 3)?
4. Can new features be created using known signatures for each specific family (goal 3)?
5. How to perform automatic feature selection to filter features with high discriminative power (goal 4)?

This concludes Part 1: Problem Analysis. Part 2 will deal with the Technical Analysis, where the aforementioned sub-problems will be discussed and answered. It includes two chapters, Chapter 5 which deals with the technical analysis of malware detection and choice of families. The next chapter, Chapter 6 deals with the technical analysis of the pre-processing of the data.

Part II

Technical Analysis

INTENTIONALLY LEFT BLANK

CHAPTER 5

DETECTION & CLASSIFICATION

This chapter includes a presentation of the existing framework and the improvements needed for this study. Then, the analysis system along with the Machine Learning toolbox used in this project, are introduced. Furthermore, a state-of-the-art research done on malware detection and classification in the scope of the Problem Statement in Section 4.1, are described.

5.1 Improving the Existing Framework

A flowchart of the existing system, is depicted in Figure 5.1, which needs to be improved in this study. The **green** highlighted boxes are the modules, where major improvements are made. It should be noted here, that a newer version of Cuckoo Sandbox is installed on the server, which will be explained in Section 5.2.

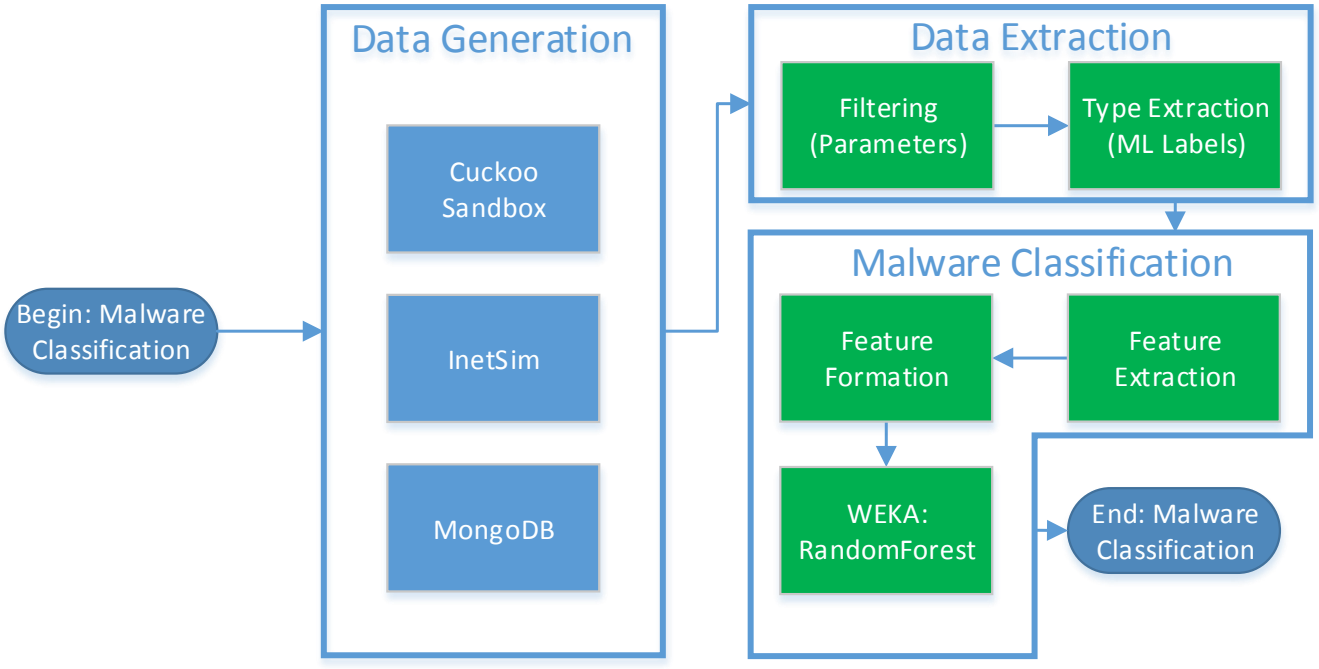


Figure 5.1: Flowchart of the previous system, see [Larsen et al., 2014].

The previous framework consists of three main groups. The first group named **Data Generation** includes the Cuckoo Sandbox module in which the dynamic analysis of the malware is performed. In the same context InetSim is a module used to emulate Internet services, so the malware believes it is on a normal machine connected to the Internet. Furthermore, the NoSQL-based database MongoDB, is used to efficiently retrieve data. The next group named **Data Extraction**, includes the pre-processing of the Big Data, and aimed to filter the noise of the data and to extract the class-labels. The last group

is named **Malware Classification** and included feature extraction and formation, together with the classification of malware.

As can be seen in Figure 5.1 (p. 19), **no** major modifications to the Data Generation module will be made. The only change here, is a software update and adding more machines to Cuckoo, in order to address the great amount of additional malware.
The next section will now describe Cuckoo Sandbox, the architecture of the analysis system and the modifications made to the Cuckoo framework.

5.2 Analysis System

This section introduces the analysis system, together with changes and improvements that have been made. Also, a cleanware analysis setup will be documented, which is different from the malware analysis environment. This is done to ensure that the two analysis setups will not interfere with each other.

5.2.1 Analysis System Architecture

The architecture of the analysis system from last study, can be found below in Figure 5.2.

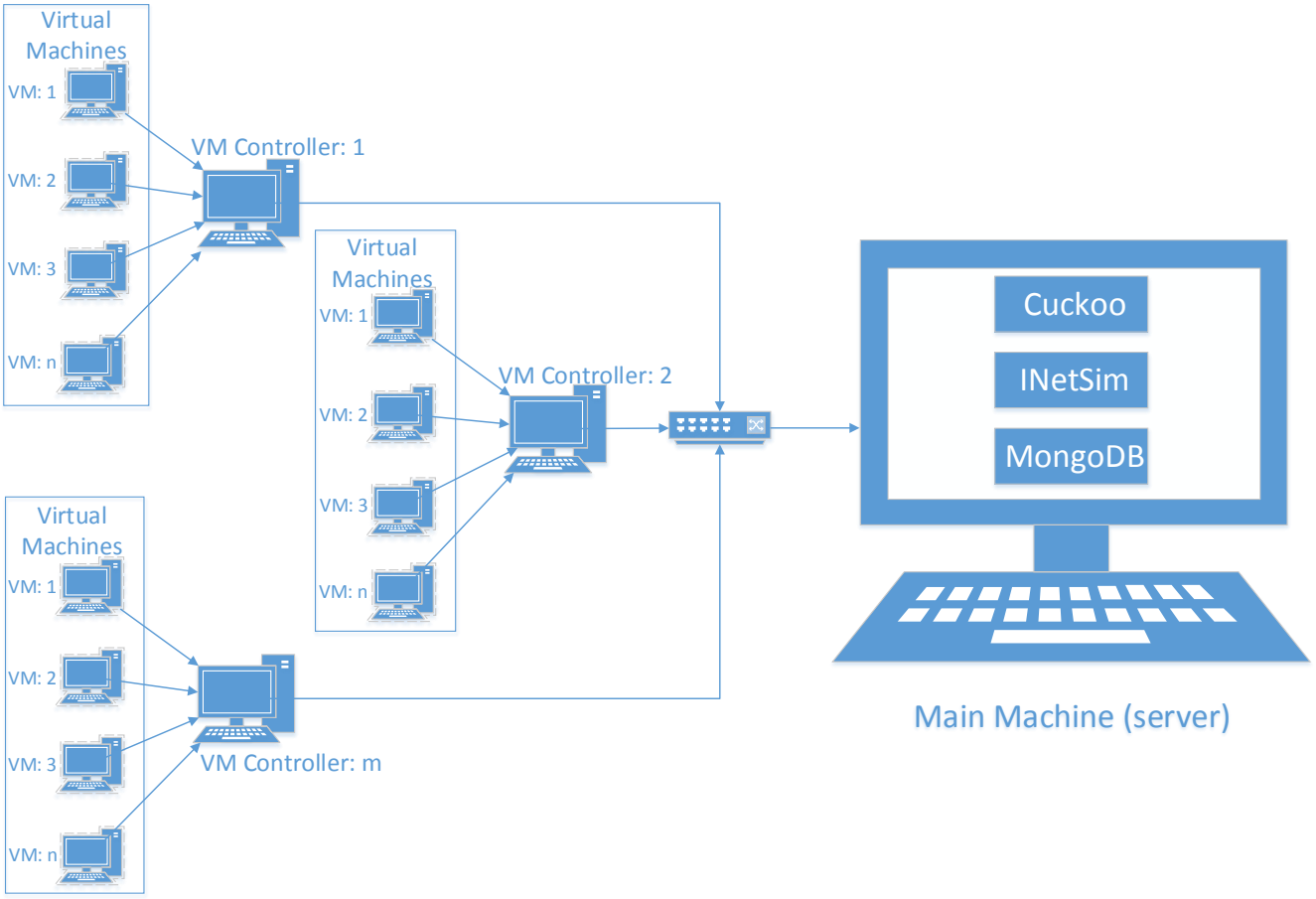


Figure 5.2: The system architecture of Cuckoo Sandbox, INetSim and MongoDB. [Larsen et al., 2014]

Cuckoo Sandbox is an open source program, developed to analyze malware. This is done by recording any

changes made to the system while the malware is running. To start an analysis, the user first queries the samples that need to be analyzed, after which Cuckoo opens a VM containing a OS, where the samples are analyzed. After opening the VM, the sample is injected, and all the changes made in the system during run-time, are recorded and stored in a Binary JavaScript Object Notation (BSON) file. Cuckoo has been modified such that the VMs run on other machines, separated from the Main Machine as depicted. This modification ensures, that the setup is scalable and that more resources are available on the Main Machine.

The VMs need to be set up by the user himself, and contain Windows 7 with no service pack, updates or Anti Virus program. To potentially maximize the activity of malware, the User Account Control (UAC) is disabled. Also, some common programs have been installed, and a browsing history have been created along with a new non-standard desktop background. This is done in order to make the malware believe, that it is not monitored. A snapshot of the current state is made, such that Cuckoo can revert after having injected a malware. By reverting to the snapshot, all traces of the malware execution are erased. This speeds up the analysis process, since reverting only takes a couple of seconds, where the time it takes to set up the system again is significantly larger. In this study, each malware sample is running for 200 seconds, after which it is terminated by Cuckoo, but if the program finishes before, Cuckoo terminates the VM. In case of a VM not responding, a critical timeout of 400 seconds has been set. If the timeout is reached the analysis is registered as failed.

None of the VMs are connected to the Internet, so in order to make the malware believe that it has Internet access, a program called INetSim is used to emulate common Internet services. It works by responding to known protocols, and provides fake files if requested by the malware. INetSim is modified so the Network Connectivity Status Indicator (NCSI) in Windows, indicates a Internet connection. This indicator is sometimes used by malware to check for Internet access. After analysis the BSON files are processed by Cuckoo, which generates JavaScript Object Notation (JSON) files, i.e. the reports that are imported to MongoDB. This makes the data easier to access for further analysis. For more details about the modifications made in Cuckoo, see [Larsen et al., 2014].

Improvements Made

More VMs have been added to the system in this study, done in order to minimize the analysis time. Furthermore, new updates for Cuckoo have been released, and in order to ensure a stable setup, the modified Cuckoo version, is updated to the newest release: version 1.2 (2015-02-XX). The *XX* was not specified in the update log. The modified version of Cuckoo can be found on the CD in `Programs\Cuckoo\CuckooMalwareSetup.zip`. INetSim has not been updated since last project and the same release is therefore used: version 1.2.5. A copy of the program together with the configuration file for the setup, can be found on the CD in `Programs\INetSim\`.

During this project, 8 VM controllers were connected to the Main Machine, but due to difference in hardware for each controller, a different number of VMs were installed. Table 5.1 (p. 22) shows the number of VMs for each controller.

VM controller ID:	PC1	PC2	PC3	PC4	PC5	PC6	PC7	PC8	Total VMs
Nr. of VMs	4	8	4	2	2	2	4	4	30

Table 5.1: List of VMs for each controller.

Ubuntu 12.04 has is installed on each controller, in which VirtualBox 4.3.24, hosts the specified number of VMs. Each VM has a bridges connection to the network card, such that VirtualBox emulates the Network Interface Controller (NIC). The Main Machine is the only computer with Internet connection, which is needed for DNS request and VirusTotal statistics. It has two NICs, one that is connected to the VM controllers through a switch, and another NIC for the Internet connection. The used malware is only Windows compatible and should therefore not be able to escape the environment and infect the Main Machine. All network details associated with Cuckoo and INetSim, can be found in Appendix B (p. 127). Their configuration files, can be found in their respective folders on the CD `Programs\`.

Based on previous experience, it was found that the Read/Write (R/W) speed of the HDDs, was a bottleneck during the analysis. To ensure the efficiency of the computer hardware, all machines connected to the analysis system, including the Main Machine, was installed with two HDDs in RAID 0. This increased the R/W speed, but with the downside of a higher risk of failure or corrupted data. This is not a big problem for the VM controllers, but mostly for the Main Machine, since all analysis data are stored here. A backup of the main machine is done for restoration if a HDD fails.

Using RAID 0 on the the Main Machine, ensures high performance, which is needed when storing the analysis data from 30 VMs in real-time. For performance, processing of the raw recorded data, is done after the analysis.

5.2.2 Cleanware Setup

The analysis setup for cleanware, uses the same components as the analysis system for malware, with the exception of MongoDB. The setup only contains 7 VMs, all located on the same machine as Cuckoo and INetSim. The machine is powerful enough to perform analysis on all 7 VMs, and therefore it is not necessary to use the distributed approach. This was also sufficient considering the lower amount of cleanware sample. The analysis setup, also uses a slightly newer release of Cuckoo: version 1.2 (2015-03-04), which only concerns the stability of the program. This does not have an impact on the results.

Cleanware and malware are analyzed differently, and the reason why, will be elaborated later in the report, see Section 5.4. Since the cleanware is located on the VMs before execution, Cuckoo has to execute the program in its location. To solve this, a shortcut pointing to the destination, is injected as the sample file. The Cuckoo program and configuration can be found on the CD in `Programs\Cuckoo\CuckooCleanwareSetup\`.

5.2.3 MongoDB

To ensure that the analysis data is easily accessible in a efficient manner, it was chosen to use MongoDB, see [Larsen et al., 2014]. The JSON files provided by Cuckoo Sandbox need to be less than 16 MB before

it can be imported to MongoDB. Therefore it is necessary to minify each JSON file, since some of the JSON files are above the limit of 16 MB. The script for minifying and importing JSON files, can be found on the CD in `Code\DataMangement\` as `minifyncopy.bash` and `insertdb.bash`.

The analysis data from the last study, is imported to the same collection as the new malware data. Additionally, the analysis data for cleanware, are imported to MonogoDB in a different collection. As the DB contains roughly 1.7 TB of unstructured text-based data in relation to this study, it has to be restructured in a new collection. This collection contains the data that will be used for machine learning. This includes **feature extraction**, where potential features are extracted from the DB, and **feature selection** which in this project will rely on algorithms to select the most discriminative features, see Section 7.1. When the data has been restructured, the training and testing stages can begin. To understand the structure of the database, the generated JSON file from Cuckoo is described next.

5.2.4 JSON Structure

This section will present the database structure including the objects that are used in this project. The tree in Figure 5.3 shows nested objects of the parent objects **behavior** and **virustotal**. The collection, which is the direct import from Cuckoo, is called `analysis`. It includes all the information used in this project on a per sample basis. This means that for each injected sample, the depicted database structure is used. Note that if the whole database structure was shown it would take up 11 pages, which is why it is not shown here.

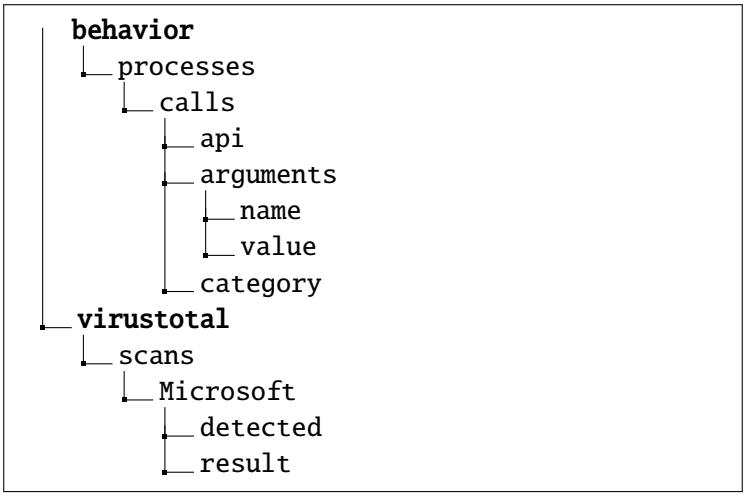


Figure 5.3: The unchanged JSON structure.

Using these objects, behavioral features are extracted along with Microsoft labels for the classification. During the project, additional smaller collections are created including information extracted from the `analysis` collection.

This section described how the analysis setup works and how data is retrieved and stored. This can now be used for further analysis in detection and family classification. The next section presents the WEKA Machine Learning toolbox.

5.3 Machine Learning: WEKA

This section describes the Waitkato Environment for Knowledge Analysis (WEKA) Machine Learning toolbox used in this project. This will be done with emphasis on the functionalities this study plans to use, hence it will not describe all its functionalities.

There is a wide range of functionalities for pre-processing the data e.g.: attribute selection, visualization, clustering and classification. In general, the tool provides a large collection of Machine Learning (ML) algorithms, which can be applied to a corresponding data set after it has been converted to Attribute-Relation File Format (ARFF). The functionalities used in this study will now be described.

5.3.1 Selecting Features

As mentioned in Subsection 4.1.1, one of the sub-problems that has to be solved, are to use a feature selection algorithm. WEKA fortunately supports e.g.: Correlation-based, Gain-based, Information-gain-based (entropy), etc.. The aforementioned evaluators utilize specific search algorithm, e.g. Best First, Ranker, Re-ranking-search, etc.. These algorithms will be chosen later in Section 7.1.

5.3.2 Classifiers

There are a wide range of classifier groups that can be used in WEKA like: Bayes, Lazy, Meta, Rules, Trees, etc.. If the data needs to be pre-processed first, this is easily done using the aforementioned algorithms, where WEKA can remove the filtered features and prepare it for the classification.

Based on the tests made last semester, it has been **decided to use Random Forests**, which was the algorithm that showed best results. This algorithm will be further explained in Section 7.4. The number of trees to be used will be found in the Training Stage, see Chapter 10. The next section describes Malware Detection.

5.4 Malware Detection

This section include potential methods for solving the first sub-problem listed in Subsection 4.1.1:

- How to perform malware detection?

The presented methods can be used to detect malware using certain parameters from the analysis data. The goal for detecting malware is the ability to discriminate cleanware and malware using run-time behavior. The definition of cleanware was described in Subsection 3.1.1. Detecting malware without the need of signatures, is useful especially if combined with the last project. This is furthermore supported by the fact that some AV vendors are relatively slow in updating the malware signatures, [Salehi et al., 2012]. In relation to this statement the method for detecting malware presented here, will not depend on signatures as many AV-vendors do. Detecting and classifying malware in a dynamic analysis perspective, can potentially help AV-programs to detect unknown and zero-day malware. To illustrate how the system could work in a final state, the following flowchart in Figure 5.4 (p. 25) is depicted.

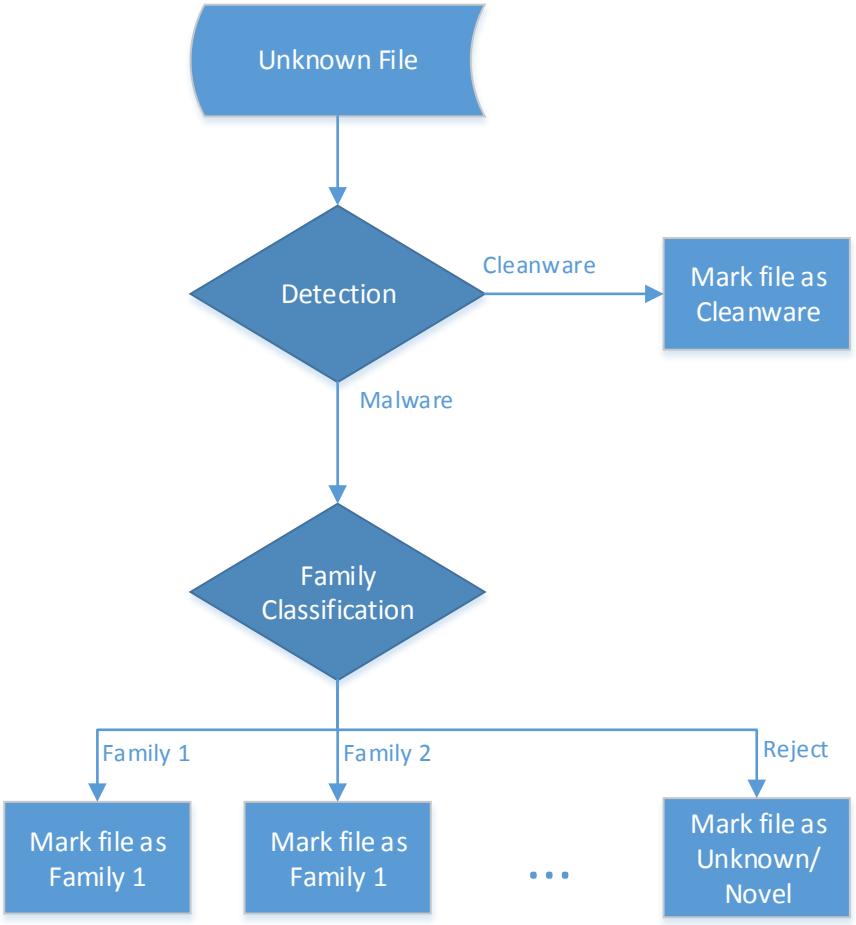


Figure 5.4: Illustration of the framework in a final state.

When malware is detected, the family classifier specifies the family it belongs to, but should also be able to reject if the probability of the classification is too low.

The next subsection will examine the methods of malware detection, after which the method used in family classification is presented.

5.4.1 Discriminating Malware & Cleanware

The foundation for the last project, was that certain parameters recorded during execution, such as API calls, can be used to make a run-time behavioral profile of each malware type. This worked and classification of different types was carried out. The same idea should work for cleanware, assuming that the run-time behavior differs from malware. Due to this assumption, cleanware will be limited to installed and portable applications on the VMs, where only the run-time behavior is recorded. Future work can expand the method, such that detection can be used for the installation process, and files such as .doc, .pdf and etc..

The reason for installing the cleanware first is due to limitations of Cuckoo, that makes it hard to automate installation and execution of programs. The first limitation is that the installation process did not always complete, since Cuckoo stops pressing next, which results in an installation that never starts.

The second limitation, is that Cuckoo does not automatically open the program after installation. This feature is normally not needed since malware is fully automated after execution. Solving these issues will still pose a problem when having to include the data from the installation process. This is because of the 16 MB limit when importing JSON files to MongoDB. Since the installation process generates more data, exceeding the 16 MB limit even after minification, it cannot be imported.

It is assumed that malware will exhibit primarily run-time behavior, whereas the amount of malware which might show a installation process similar to cleanware, is small. This is verified when looking at the malware run-time data, which is reasonable since malware wants to hide itself, to avoid detection. Though, "installation" of malware could involve changing reg. keys and copying itself to another directory, for instance to ensure that it is executed during startup. This "installation" process is negligible compared to installation of most cleanware programs, hence, the project will only consider cleanware run-time behavior.

Since cleanware is pre-installed on the VM or located in a folder as portable programs with dependencies, it is needed to execute the cleanware using shortcut files. If only the .exe file is injected, an error will occur because it is dependent on other files. From Section 3.4.1 it was seen that some programs are installed, some are portables and other are Windows native programs. The shortcuts for all installed programs, are extracted from the start folder, by searching for .lnk files. The shortcuts for the portable programs need to be created first using a script, which can be found on the CD in folder `Code\ShortcutGenerator\`. To run the analysis, the shortcuts are queried and during the extraction of the analysis data, all data associated with the shortcut can be removed. This is needed, since it has nothing to do with the behavior of the cleanware.

Now that it is clear how, and which kind of cleanware programs, that will be injected, it is believed that malware detection can be done using the same the method as for family classification. This assumption is believed to be true if an additional class is made for cleanware by:

1. Creating one class which includes all behavioral categories for cleanware - binary classification.

The binary classifier with will include one cleanware class and one malware class, which will be described below.

Binary Classifier

This method assumes that a classifier can discriminate between cleanware and malware. If the assumption holds for having one cleanware class, malware detection based on dynamic analysis, could be possible. This method is depicted in Figure 5.5 (p. 27), where an unknown file is passed to a binary classifier.

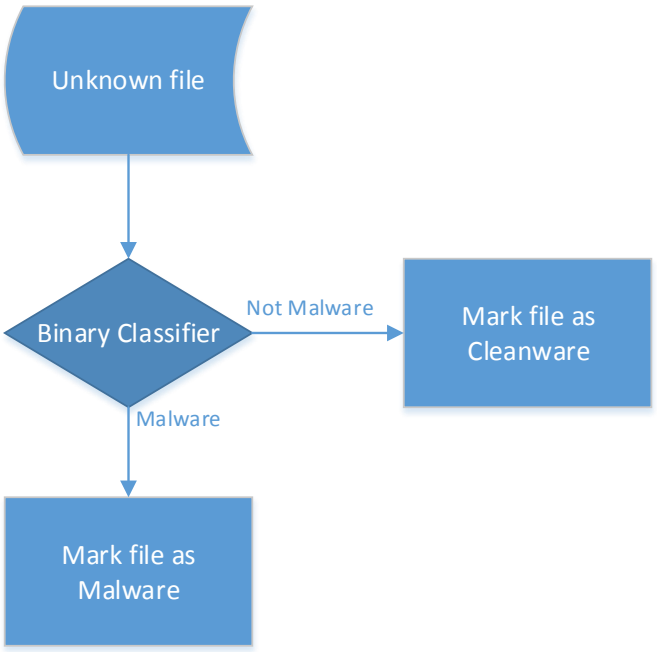


Figure 5.5: Illustration of the method using a binary classifier.

The presented method is inspired and also used by different papers, in which the method has given promising results. One of them is [Tian et al., 2010], in which a smaller number of malware namely 1,368 and 456 cleanware files, were analyzed by dynamic analysis. With their feature representation they managed to achieve an accuracy of 97.3 %. They also presented a family classifier, which had an accuracy of 97.4 %. Here the classifier was made based on the 1,368 malware retrieved from 10 different families. Also, other papers like [Uppal et al., 2014], [Salehi et al., 2012], [Shahzad and Lavesson, 2011] and [Firdausi et al., 2010] use dynamic analysis with a relatively small number of malware and cleanware files, mostly below 1,000 in total, and achieved satisfactory results. It should be noted that [Shahzad and Lavesson, 2011] use static analysis, if the sample does not exhibit any run-time behavior. [Kasama et al., 2012] developed a system that uses dynamic analysis for malware detection, with the goal of achieving a low False Positive Rate (FPR). Here they detected 67 % of the malware, but in return only had 1 % FPR. The same method is used in [Fukushima et al., 2010], and they were able to achieve a True Positvie Rate (TPR) of 60 % for malware with no false positives. Though, their sample set was small, with only 83 malware and 41 cleanware files.
Other papers used static analysis to collect parameters for malware and cleanware files also supporting a binary classifier, which can be found in [Santos et al., 2011], [Vinod et al., 2012],
[Zolotukhin and Hamalainen, 2014] and [Saini et al., 2014]. They all achieve promising results using around 5,000 files and below. [Markel and Bilzor, 2014] uses static analysis, but here they utilize a much larger sample set with 122,799 malware and 42,003 cleanware files. At last, one paper [Liu et al., 2013], uses a hybrid, i.e. a combination of both dynamic and static analysis, to extract different kind of features and achieved satisfactory results.

The different amount of cleanware and malware used in the different papers are listed in Table 5.2 (p. 28). Even though many papers have done this before and the majority ended out with good results, few

Analysis Type	**Number of Malware**	**Number of Cleanware**
Dynamic [Tian et al., 2010]	1,368	456
Dynamic [Uppal et al., 2014]	120	150
Dynamic [Salehi et al., 2012]	826	385
Dynamic [Shahzad and Lavesson, 2011]	550 (Scareware)	250
Dynamic [Firdausi et al., 2010]	220	250
Dynamic [Kasama et al., 2012]	5,697	819
Dynamic [Fukushima et al., 2010]	83	41
Static [Santos et al., 2011]	1,000	1,000
Static [Vinod et al., 2012]	4,805	2,828
Static [Zolotukhin and Hamalainen, 2014]	47	953
Static [Saini et al., 2014]	2,460	627
Static [Markel and Bilzor, 2014]	122,799	42,003
Hybrid [Liu et al., 2013]	11,378	1,712

Table 5.2: Summary of the amount of malware/cleanware samples in the papers.

papers have tested this method with a large sample set. Only [Markel and Bilzor, 2014] used a relatively large sample set, but they did not collect the parameters through dynamic analysis, but instead used static analysis. Also some papers did not specify if the cleanware was executables, whereas most of the cleanware used in the papers native Windows programs.

This study uses dynamic analysis to retrieve other kinds of feature parameters, which will be described later in Section 6.1. The features and the representation technique are different compared to the ones proposed in the papers, which will be described in Section 6.3. The next section will describe the families and decide which to use.

5.5 Malware Family Classification

The goal of this section is to solve the second sub-problem listed in Subsection 4.1.1 which was:

- Which malware families should be used as class-labels?

The next paragraph will discuss the family classification and which families to use as proof-of-concept.

5.5.1 Malware Families

After the malware detection presented in Section 3.1 a second classification will be executed in the case, where the injected program was determined to be malicious. It was chosen to perform family classification in Section 3.2, where the benefits of choosing malware types and families were discussed. The choices are made on the following requirements:

- The family must include a significant amount of samples, while maintaining uniformity among the classes.

- The family must have existing detailed descriptions of potential actions, see Subsection 6.3.3 for an explanation of why this requirement is needed.

One of the goals in this study is to achieve a sample set distributed in a uniformly manner among the classes. For this reason, the largest malware families have been taken into account. Before the families are chosen, detailed descriptions of their known potential actions is checked. The descriptions have been found on professional AV-vendor websites. It is well known that the AV-vendors does not name the families using a common standard, but searching for family aliases have proved to be possible on several websites. These websites include the following: [Microsoft, 2014], [Trendmicro, 2015], [Symantec, 2015] and [F-Secure, 2015].

Studying the distribution of malware families in the DB, see Figure 5.6, it is seen that the top five have above 5,000 samples. Note here, that it was chosen to use Microsoft's labeling in Section 3.3, since they use a standardized naming scheme called CARO, see Subsection 3.3.1.

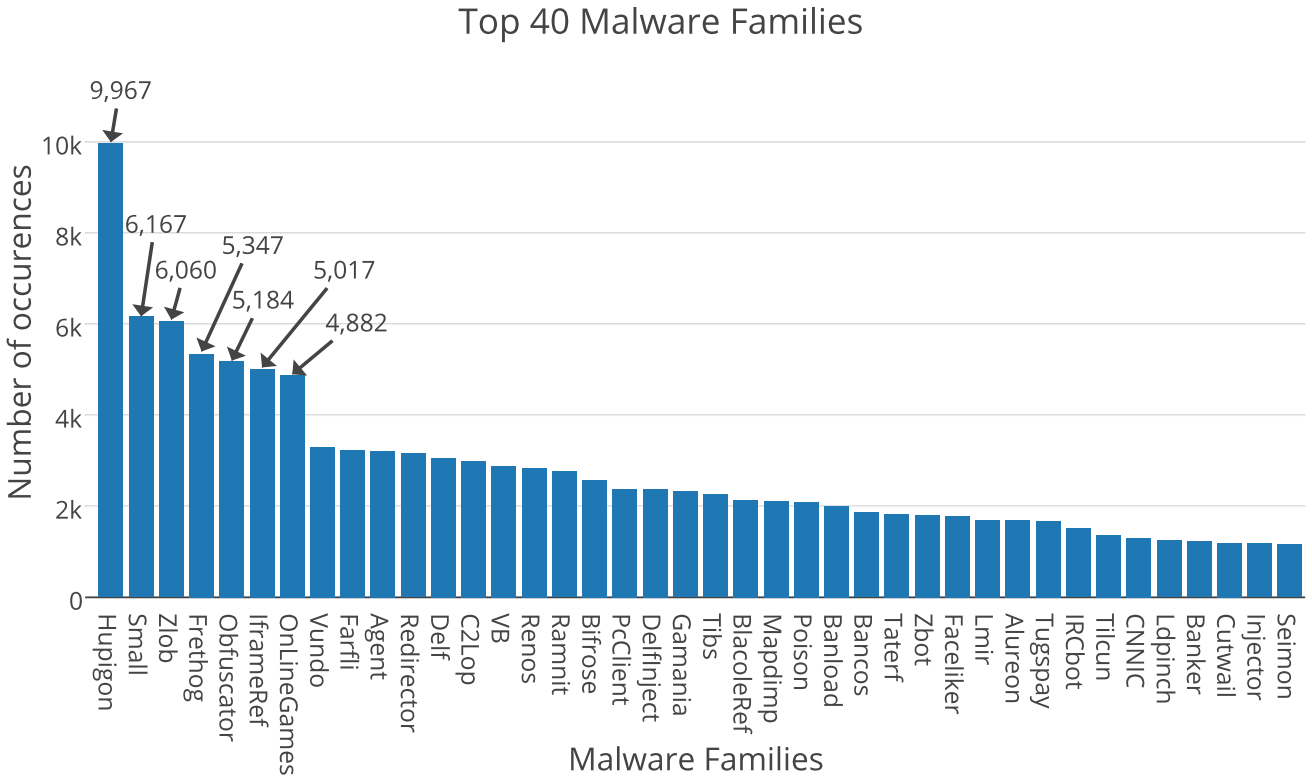


Figure 5.6: Distribution of malware families labeled my Microsoft.

In addition, this graph has been cut off to show the top 40 families. The full list includes 6,789 distinct families, in which all families after the top 215 families had below 100 samples. This observation is important to notice because it indicates, that there exists a wide range of malware families, which could become problematic if all are used. Using all families would require a significant amount of samples for each. Because this project is proof-of-concept, only five families will be taken into account. This is due to the hardware limitations, but it is assumed that if implemented in a company with sufficient resources, the classification will work. Furthermore, it should be noted that for malware detection all families will

be used. Shown in Table 5.3 are the 7 largest malware families in the DB together with the exact amount of samples and a check mark to indicate if detailed descriptions were available.

Families	Amount of samples	Detailed description
Hupigon	9967	✓
Small	6167	✓
Zlob	6060	✓
Frethog	5347	✓
Obfuscator	5184	✗
IframeRef	5017	✗
OnLineGames	4882	✓

Table 5.3: List of malware families named by Microsoft and their amount along with description check.

As can be seen, **Obfuscator** and **IframeRef** did not pass the description check. This is due to the vague malware descriptions by Microsoft. They describe **Obfuscator** as a family that tries to hide its purpose in order to evade detection and they mention that the purpose of this malware can be almost anything, see [Microsoft, 2014]. The description indicates a behavior and not a family originating from the same source, hence it will not be used for classification. The same goes for **IframeRef**, which in most cases are JavaScript-based malware, which redirects the browser to another website, see [Microsoft, 2014].

The next paragraphs will briefly describe what the chosen families are commonly known for based on Microsoft's threat encyclopedia. Here, the goal is to understand the classes when determining what features or signatures could be relevant, when training the classifier, see [Microsoft, 2014].

Hupigon A large family commonly known to be a Remote Access Tool (RAT). By registering itself as a service on the compromised machine, it opens a backdoor server, where the perpetrator, through other computers, can connect and control the infected machine.

Small Known to be a family of Trojans that without the users content, connects to a remote server, where it downloads unwanted software, like additional Trojans or imitated security programs.

Zlob This malware family is very generic as it consists of multiple components like: modification of Internet Explorer's settings, altering of the user's default Internet search page and home page and also tries to download and run other malicious programs. It might also display warnings regarding untrue malware infections.

Frethog Known to be a large family, where the main focus is to steal passwords to access confidential data like account information from for instance the massive multiplayer online games: World of Warcraft.

OnLineGames This is also a family of Trojans, but can both execute dropping and information stealing behavior, and typically collects key strokes from online games. By dropping it means that once it has infiltrated the system, it downloads more malware and drops it on the compromised machine.

From this analysis it has been chosen to use the five aforementioned families as class labels. Note here, that the detection will include all the families since the goal is only to detect malicious activity.

5.6 Chapter Summary

This chapter first presented an overview of the existing framework and analysis system, in relation to the present study. Furthermore, a ML toolbox were presented together with theory and choices, based on the goals of this project. This included the malware detection approach, classification approach and what families to include. The conclusions in this chapter will be the basis for the further analysis of the technical aspects in Chapter 6. The main conclusions are summarized in the following:

1. **Improving Existing Framework:**
   — Included a summary of the existing framework.
2. **Analysis system:**
   — Included a summary the analysis setup along with technical details about the improvements that are made.
3. **Machine Learning: Weka:**
   — This section presented WEKA ML toolbox, where Random Forests were chosen as classifier.
4. **Malware Detection:**
   — Here, a binary classifier was chosen for malware detection.
5. **Family Classification:**
   — Here, the following families were chosen for classification: Hupigon, Small, Zlob, Frethog and OnLineGames.

The next chapter presents what Windows parameters to use as features along with a presentation of how the data should be pre-processed. This is important, as it is beneficial for the predictive performance of the classification in both malware detection and family classification.

INTENTIONALLY LEFT BLANK

CHAPTER 6

PRE-PROCESSING OF DATA

This chapter presents the first stage in developing the classification, namely the pre-processing of the data. It firstly includes an analysis of potential features followed by the feature extraction approach along with how the features are represented.

6.1 Windows Parameters

This section deals with the fourth sub-problem in Subsection 4.1.1 which was:

- Can new features be created using DLL, Mutexes and I/O arguments for API?

This section goes into details of different Windows parameters that are recorded during the analysis. The different parameters are the following: Dynamic-Link Library (DLL), Registry (REG) key, mutex and Windows Application Programming Interface (API). Below, each parameter is explained, starting with its normal procedure, how malware can use the parameter and at last how it can be used when analyzing the malware. At the end of this section the features used for this project will be chosen.

6.1.1 Dynamic-Link Library

A Windows DLL file is executable code in form of a library, which can be loaded by a program, either at startup or when needed. The DLL files provide needed functions, and it is therefore common to see programs rely on many different DLL files. One DLL file can be used by multiple programs simultaneously, which thereby save memory resources, see [Microsoft, 2015g].

Analyzing the behavior summaries in Cuckoo, information from the malware can be found by evaluating the used DLL. This is because the DLL and its functions, can point to the behavior of the program. A problem arisen in the recent years, is malware that can hijack DLLs and inject their malicious code. This means, that the malicious DLL file is used instead of the original, see [Evans, 2011]. From the analysis, the potential created or modified DLL files can be used to define the behavior of the malware, assuming that the name of the DLL is not random. This will be the case if it hijacks a Windows DLL, since the name has to be identical. To conclude, DLL files can be used to form a behavioral profile of the malware, and should therefore be considered as features.

6.1.2 Windows Registry

In Windows, the REG keys forms a database, which contains information about the user, system, programs etc.. The keys contain values, that are used to describe the particular setting of that key, see [Microsoft, 2015h] and [Fisher, 2015]. Malware can use REG keys to store its malicious code, see [Santros, 2014], or to ensure that it is executed at Windows startup. The benefit of this, is that it is

possible to use them for indicating its behavior, with the exception of cases where the REG keys are randomly named. In other cases behavior can be extracted by looking at those accessed during the analysis, since it gives the malware information about the system, programs and users.

6.1.3 Mutex

Mutexes are used in threading, where multiple threads are sharing resources. In order to ensure, that two threads are not using the same resource simultaneously, mutexes are used. This is accomplished by changing the state of the mutex to "nonsignaled" when a thread is using the resource, i.e. "nonsignaling" means that the mutex is currently owned by the thread. Since a mutex can only be owned by one thread at a time, it is ensured that no other thread can access the resource. After having used the resource, the mutex is released, and another thread can claim ownership of it. In other words, a mutex can be seen as a lock, see [Microsoft, 2015c].

Malware sometimes use mutexes to ensure that the compromised machine is not infected with another version of the same malware. Here, it tries to claim ownership of a mutex, which will be impossible if the machine is already infected by the same malware. If the malware fails in claiming ownership it will delete itself. It is also believed that some malware tries to create mutexes, that are associated with VM programs. If it fails in claiming ownership of such mutex, it believes that it is being monitored in a VM environment. This technique could also be used to hide the malicious activity, by creating a mutex with a unsuspicious name, see [Blasco, 2009].

Using the mutexes utilized during the analysis, it should be possible to use names associated with the malware. This could potentially help discriminating the particular malware, as this can become a signature of the malware. This is of course only possible, if the names of the mutexes used, or created by the malware, are not randomly generated. This should not be the case, if the mutex was created to check if the machine has been infected with another version of itself.

6.1.4 Windows Application Programming Interface

An API is used to interact between different software or hardware modules. Here the API specifies a set of functions that act like building blocks, which are independent of how the functions are implemented, see [Wikipedia, 2015a]. In this project, Windows APIs are used, which e.g. makes it possible to access lower level functions, such as creating a file, but also higher level applications, see [Beal, 2015], [Microsoft, 2015b] and [Franklin and Coustan, 2015]. One of the reasons why Windows use APIs, are to make it possible to code applications that will work across all versions of Windows, see [Microsoft, 2015a]. From the above and the paper [Alazab et al., 2010], it is derived that one can use API calls to resolve the behavior of an application like e.g. a Portable Executable (PE). This is because, APIs are connected to a functionality, and by using the different API calls, it is possible to exploit the information by developing a behavioral profile of the application.

It is known that malware types can be discriminated using different API features, and this should also be possible for cleanware. Furthermore, the input arguments for each API call, contain information about

the specific operation. The argument can specify details about, which DLL file is loaded, mutex created/modified, REG key accessed and etc.. This is because, different API calls are used to handle these parameters. It is therefore assumed that using the input arguments for the API calls, can reveal more information about the behavior compared to the name of the API, which was shown in [Salehi et al., 2012]. It is assumed that the arguments can improve the classification. This is further analyzed in Subsection 6.2.1, where a new method of extracting information from the DB is presented.
The number of occurrences for each API, for both the malware and cleanware analysis, can be found in Appendix E and Appendix F, respectively.

6.1.5 Choice of Parameter

It is decided to use **API calls with input arguments** in this project, since all API calls in Cuckoo are strictly related to the executed file executed, tracked by its PIDs. Note, that a file can have multiple PIDs, but they will all point to their PIDs. The input arguments, ensure that parameters as DLL, mutexes and REG keys can be used in the detection and family classification.

6.2 Feature Extraction

This section explains why it was necessary to "whitelist" the data output from Cuckoo in the previous study. This was done, because Cuckoo records all actions done on the OS, even though they were initialized by non-malicious software or the OS itself. Everything, except the API calls had no pointer to the PIDs of the malware program. Therefore, it was reasonable to make a "white experiment", in which a legit bash script was executed in Cuckoo. This experiment was used to filter the information in the behavior reports (except API), that was also present in the "white experiment". Here, the assumption was, that it would remove the majority of the noisy data created by legit operations on the compromised machine.

The following paragraph discusses an approach to extract features in a more sophisticated manner, since it is believed that the "whitelist" created in the last study, would potentially filter out information that was generated by the malware. This could for instance be, a path accessed by the malware, which was also accessed by the OS.

6.2.1 API Arguments

Not only does Cuckoo record the API calls related to the PIDs of the malware, it also records the input arguments. Using the APIs together with the argument values, it is possible to track all the actions performed by the malware. This not only include the used APIs, but also the paths, mutexes and REG keys accessed or modified during the experiment. Knowing this, it is no longer necessary to rely on the *Behavior Summary*, since this information can be extracted from the arguments. An example of a DB query, is seen in Code snippet 6.1, where the arguments are objects in a list.

Code snippet 6.1: Example of API arguments as seen in MongoDB.

```
1                    {"category" : "services",
```

```
                        "status" : true,
                        "return" : "0x001b7848",
                        "timestamp" : "2015-02-19 18:03:09,407",
                        "thread_id" : "3816",
                        "repeated" : 0,
                        "api" : "CreateServiceA",
                        "arguments" : [
                                {
                                        "name" : "DisplayName",
                                        "value" : "GrayPigeon_Hacker.com.cn"
                                },
                                {
                                        "name" : "BinaryPathName",
                                        "value" : "C:\\Windows\\Hacker.com.cn.exe"
                                }
                        ],
                        "id" : 46
                }
```

This example shows arguments for the malware family Hupigon. This also explains why it creates a new service, as this is a common trait of this family, see the family description in Subsection 5.5.1. Depending on the amount of arguments specified for the particular API `CreateServiceA`, the list can have multiple entries. It is noticed here, that the argument value for `BinaryPathName`, is the specific path `C:\\Windows\\Hacker.com.cn.exe`, in which it tries to create a service. Extracting features from the API arguments, are assumed to contain better information for discrimination. The process are the following:

1. Create list of regular expressions to extract the needed information to a new collection.
2. Use the remaining data as potential features.

Based on the decisions made in Subsection 6.1.5, regular expressions can be used when extracting the needed information. More in depth explanation of this will be described in the Design & Implementation, see Part III.

6.3 Feature Representation

This section examines the feature representations that are used in this study. The representations or representation techniques: binary, frequency and sequence, will be the same, but the features they represent, differ from the last project [Larsen et al., 2014]. The terms and the representation techniques will be elaborated in this section.

From Subsection 6.1.5, the features were chosen to be API calls and some of their respective input arguments. In the last study, it was discovered that the API call sequence gave promising results. It will be used again in this project, but instead, concatenated with the some of the API input arguments. Furthermore, the frequency of the API calls, i.e. the occurrences of each specific API, will be used. Because this project utilizes the input arguments from the APIs, it is assumed that better results can be achieved

by introducing different forms of features, represented by their frequency. Furthermore, a binary feature representation is used to incorporate a signature check for each family, i.e. is the signature present or not.

Both general and detailed features are needed, after which the feature selection is used to pick the best features, based on the chosen relevance measure, see Section 7.1. This is also why a greater number of features will be listed here, as those which are not relevant will be removed.

Below, each representation technique is presented together with the features they represent. After all feature representation techniques have been presented, the final feature representations used for malware detection and family classification, will be shown.

6.3.1 Sequence Representation: API Calls with specified Input Arguments

From the last study, it was seen that features based on the sequence of APIs from execution start, could produce good results. Here, a sequence feature is defined by its unique name, e.g. the name of the API call, and will therefore also be used a nominal feature. The first API will therefore be the first feature, and the second API, the second feature and etc.. An illustration of the concept can be seen in Equation 6.1, which is from the previous study, see [Larsen et al., 2014].

$$API_{seq} = \begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_m \end{array} \begin{array}{c} \begin{array}{cccc} SEQ_1 & SEQ_2 & \cdots & SEQ_n \end{array} \\ \begin{bmatrix} API_{14} & API_{12} & \cdots & API_{112} \\ API_{76} & API_{47} & \cdots & API_{146} \\ \vdots & \vdots & \ddots & \vdots \\ API_{97} & API_3 & \cdots & API_{32} \end{bmatrix} \end{array} \quad [\cdot] \quad (6.1)$$

Here each row corresponds to a sample, whereas each column represents a specific sequence number.

This project extends the representation by adding additional information to the feature. Each API from the analysis data include arguments, thus in order to achieve greater detail from the API, a specified argument is included. Not all input arguments can be used, since some of them do not provide useful information in relation to Machine Learning. For this reason, it has been decided to use input arguments that point towards other functions. Analyzing the arguments, `FunctionName` was chosen. Note here, that this is a variable name and not the actual argument value. Not all APIs include this input argument, but it is one of the only arguments, which in most cases include values, that are not system paths or files. Both system paths and file creations, are avoided, since Malware can create random files or directories, making the feature less informative for a ML algorithm. `FunctionName` does in most cases specify another API/function, and is therefore assumed to be less likely to contain random data. Other kind of arguments for APIs could be function addresses or handles, which are not used in this project.

This study will use 200 sequence features. The input arguments can be added to the sequence representation in different ways, either as features itself or appended with the API feature. Both methods for feature representations are listed below:

1. Combine API and its input argument into one feature.
2. Have API and its input argument as separate features.

Both methods are presented below, followed by a decision of what to use in this study.

Sequence with Combined Features

To incorporate the API input arguments into the sequence representation, the features could be combined, thus still follow the representation as Equation 6.1 (p. 37). This is believed to provide less, but more detailed features for discriminating the classes.

Combining the APIs with its input arguments, will be done by assigning ID number to both. Assigning ID numbers makes it easier to look for trends in the features. It will not make a difference in relation to the ML algorithm if an ID number is assigned or the actual API or argument string. The 3,500 most used arguments will be used for malware and the 2,000 most used for cleanware. The reason for this, is to ensure that enough features will be present in the representation. If an argument, that is only present once, among all the samples, is used, the feature that this will create, will be less likely to discriminate the classes. The reason, that malware have been appointed more arguments than cleanware, is because it includes more samples and thereby more arguments. All 5,500 arguments from both malware and cleanware are assigned with a ID number. If two arguments are present, one of them will be removed. These duplicates will be present since the two lists were created independently, thus an argument for malware might also be present for cleanware. By removing the duplicates a slightly less amount of arguments are expected.

When both the APIs and the input arguments have been assigned an ID number, they are combined in sequence by a "-", without quotes. If the API has no input arguments or if the argument cannot be found in the list, the API will be combined with a "0" instead of an argument ID. Below in Figure 6.1, an illustration can be seen.

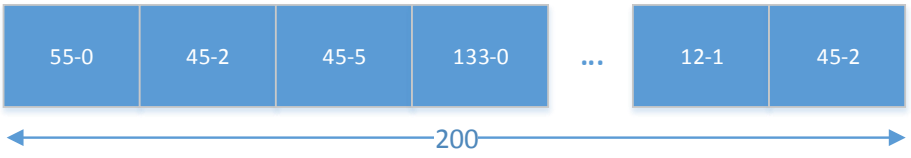


Figure 6.1: Combined sequence.

It should be noticed, that this representation will produce 200 features, as both API and its input arguments are combined into one feature. For applications that does not call 200 APIs, the represented features will include 0's on both the place of API and its arguments. The ID numbers for the APIs can be found in Appendix C.

Sequence with Separate Features

Including the APIs' input arguments as separate features will double the amount of features related to the sequence. This could be an advantage, if situations exist, where the input argument acts as a distortion,

thus leaving the feature as a whole, less discriminative. This could happen for the combined sequence presented before.
For this representation, the features will not be presented with ID numbers, but instead with the name of the feature. Furthermore, all input arguments will be utilized in comparison to before, where it was limited to $\approx$ 5,500. This might yield more details of the sample's behavior.

In order to make the representation more intuitive, the feature names (not the values) will follow the order API 1, argument 1 for API 1, etc. Below in Figure 6.2, the representation for sequence with separate features is depicted.

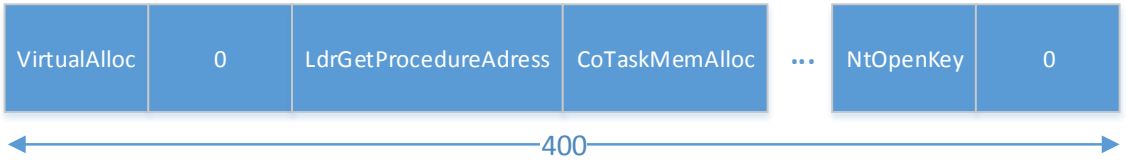


Figure 6.2: Sequence with separated API and arguments.

Notice, that the difference lies in presenting the features with their actual string value names in addition to the fact that all input arguments can be used in the representation. The length of the representation will be 400 features, but that does not matter, since feature selection will remove the redundant data, see Section 7.1.

Both presented representation methods, will be used in training and testing. This means that there will be two different representations when combined with the other representation techniques, as will be explained later. The reason for testing both sequence representation techniques, is that both have pros and cons, and one might perform better depending on which kind of classification is used.

6.3.2 Frequency Representation: API calls and input arguments

This study also addresses a representation, where the frequency of the different APIs, are used. An illustration of the representation can be seen below in Equation 6.3 (p. 41). This was also used in the last study, see [Larsen et al., 2014].

$$API_{freq} = \begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_m \end{array} \begin{array}{c} \begin{array}{cccc} API_1 & API_2 & \cdots & API_n \end{array} \\ \begin{bmatrix} 677 & 43 & \cdots & 83 \\ 4 & 8785 & \cdots & 51 \\ \vdots & \vdots & \ddots & \vdots \\ 414 & 27 & \cdots & 7397 \end{bmatrix} \end{array} \quad [\cdot] \quad (6.2)$$

Every column represents a specific API, whereas the rows are samples. Each entry then represents the occurrence of a specific API during the analysis. This representation showed to provide good results for classification of types, and is also believed to be useful for this project, in both family classification and detection.

For this study, not only APIs will be represented by frequency, since it is assumed that more detailed information could help the classifier discriminate the classes even better. Below, a list describes additional features followed by a choice of which to use.

1. Frequency bins, including categorical API calls.
    - This method uses what is called Feature Construction [Wikipedia, 2015d]. Here, bins or groups are used to construct new and reduced features containing the information from APIs of the same category. This works by summing the API calls that are in the same category. The categories are based on the purpose of the API. This method reduces the amount of features without loosing too much predictive power.
2. Frequency of the X most used Mutexes.
    - This method creates a representation of a specified number of the most used mutexes. Each column represents a specific mutex and the entry represents the occurrences of actions performed.
3. Frequency of the X most modified/created/deleted REG keys.
    - Same idea as for mutexes, just for REG keys.
4. Frequency of the X most modified/created/deleted DLLs.
    - Same idea as for mutexes, just for DLLs.
5. Frequency of the X most modified/created/deleted files.
    - Same idea as for mutexes, just for files.

Feature 1: The behavior bins were tested in the last study, and gave good results for type classification, which is also why it is used in this project. This is based on the assumption, that bins can provide features that would be more predictive for some families, than the different APIs.

Feature 2: Frequency of a number of different mutexes will provide features that could help discriminate the families. Of course, only if the malware families use the same mutex as a way of indicating existence on the system, see [Blasco, 2009]. Since there are several thousand different mutexes in the analysis data, it would be hard to justify which mutexes that indicate the existence. Additonally, a lot of the mutexes might be random, and therefore the mutexes will not be used. However, signatures for different families will be used, in which mutexes are also included, this is explained later in Subsection 6.3.3.

Feature 3: The frequency for different numbers of REG keys, could serve as features for better discrimination. Though, since REG keys might include random key names, these will not provide any discriminative information. Of course, existing REG keys might be modified, which will be useful as a feature, but because of potential randomness, it has been decided not to use them.

Feature 4: The frequency of a number of different DLLs, is believed to yield behavioral information like API/function, see [Alazab et al., 2010]. The difference is, that DLLs provide different behavioral information compared to APIs. Note, that DLLs need to be loaded before a specific function can be

used. It is believed that the frequency of DLLs are useful discriminative features, and are therefore chosen.

Feature 5: The frequency of files could also be useful, but since it in many cases contain random names, it will be less discriminative.
From the above discussion for frequency representation, it was chosen to use the frequency of APIs, API behavior Bins and for a specified amount of DLLS. The three different features will be presented below.

Frequency API

In this study 157 different APIs are seen from the analysis data, a list can be found in Appendix C. The representation have already been shown and explained, in Equation 6.3.

Frequency Bins

In the last study, the different behavior bins were made manually, but in this project Cuckoo's own description of API bins will be used. Below in Table 6.1 a list of 14 different bins can found.

1. Device	**6. Services**	**11. Network**
2. Filesystem	**7. Socket**	**12. Windows**
3. Misc	**8. Synchronization**	**13. Hooking**
4. Process	**9. System**	**14. Anomaly**
5. Registry	**10. Threading**	

Table 6.1: Bins used in the frequency representation.

Each bin will have its own column in the representation, and for each sample, the bins, which the APIs belong to, will be counted. In total, the frequency bins will utilize 14 features.

Frequency of DLLs

It has been decided to utilize a limited number of DLLs in this project, namely the 25 most common DLLs. The approximately 15 most used DLLs for malware and 10 most used for Cleanware. If a DLL is identical for both malware and cleanware, another DLL for malware should be found, so in total 25 distinct DLLs. As for the other frequency representations, the feature denotes the columns and rows denotes the samples. Below in Equation 6.3 the representation is shown.

$$API_{freq} = \begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_m \end{array} \begin{array}{c} \begin{array}{cccc} DLL_1 & DLL_2 & \cdots & DLL_{25} \end{array} \\ \begin{bmatrix} 6 & 3 & \cdots & 0 \\ 4 & 85 & \cdots & 5 \\ \vdots & \vdots & \ddots & \vdots \\ 14 & 7 & \cdots & 87 \end{bmatrix} \end{array} [\cdot] \quad (6.3)$$

The list of DLLs can be found below in Table 6.2.

1. kernelbase.dll	**10. crytbase.dll**	**19. ntdll.dll**
2. user32.dll	**11. rrcrt4.dll**	**20. rpcrt4.dll**
3. mscoree.dll	**12. urlmon.dll**	**21. setupapi.dll**
4. advapi32.dll	**13. uxtheme.dll**	**22. propsys.dll**
5. kernel32.dll	**14. dwmapi.dll**	**23. apphelp.dll**
6. ole32.dll	**15. wininet.dll**	**24. api-ms-win-security-sddl-l1-1-0.dll**
7. ole32aut.dll	**16. gdi32.dll**	**25. rpcss.dll**
8. comctl32.dll	**17. profapi.dll**	
9. shell32.dll	**18. shlwapi.dll**	

Table 6.2: DLLs used in the frequency representation.

In total 196 features have been chosen for the frequency representation.

6.3.3 Binary Representation: Signature-based Features

This section describes the idea of signature-based features, which rely on known potential behavior of the malware families. Note here, that the term "signature" should not be confused with the Signature-based Detection, but will be used in a behavior-based classification perspective.
Due to the results from the previous study, it was decided to perform family classification, with the hope of potentially achieving higher predictive performance, described in Section 3.2. Using families, specific family signatures can be used as features. This will throughout the rest of the report be addressed as signature-based features.
The idea came from studying the detailed behavior summaries from some of the vast amount of families, that exist on the Internet. The summaries provided by AV-vendors, include information based on the their analysis and may contain specific details, from both static and dynamic analysis. As an example, this could be specific paths, in which one particular family might drop files, names it might generate when creating files, mutexes and/or changes to a registry.
Here the goal of the analysis is to pick a valid amount of signatures that, based on studying these summaries, include unique signatures for each particular malware family. The families were described in Subsection 5.5.1. It is assumed, if done correctly and if enough time is invested in finding useful signatures, they will provide high discriminative power. This assumption will hold as long as each of the signatures only will be present for one of the chosen families. This is of course the best case scenario, and a little noise might be difficult to prevent, especially when increasing the variety of samples. Furthermore, the representation of signatures will be binary, and there will be a fixed amount for each of the classes. The amount is determined by, how much valuable information can be found from the provided summaries. To illustrate the idea, the following matrix in Equation 6.4 shows the representation for this feature parameter:

$$SIG = \begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_M \end{array} \begin{array}{c} \begin{array}{cccc} sig^{(1)}_{fam_1} & sig^{(2)}_{fam_1} & \cdots & sig^{(J)}_{fam_I} \end{array} \\ \begin{bmatrix} 0 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \end{array} \quad [\cdot] \quad (6.4)$$

Here, the rows denote the samples and the columns the signature feature for a specific family.
A few example signatures for Hupigon are listed in the following:

- 6600.org
- Hacker.com.cn.exe
- BEI_ ZHU
- huaihuaitudou
- GrayPigeon
- Rejoice2007

These particular signatures are known to be used in relation to naming new files, services or mutexes. A complete list of the signatures chosen in this project for each family, can be seen in Appendix A (p. 125). Again, if enough time is invested in analyzing samples from each family, discriminate features can be found. This might contradict one of the arguments of using features based on dynamic analysis, because signatures might be easier to find based on static analysis. In future work, more time should be used to analyze the malware families to find more and better signatures.

6.4 Combining Representations

This section combines the feature representations chosen in the previous sections. It was chosen to use both sequence representations for malware detection. Furthermore, both sequence representations will be combined with the signature features, which are specific for family classification. This means that there will be a total of 4 representations, which will be tested in Part IV.

6.4.1 Representation for Malware Detection

For malware detection, Representation 1 consists of 396 features in total. It represents the sequence and the frequency representations. An illustration can be seen below in Figure 6.3.

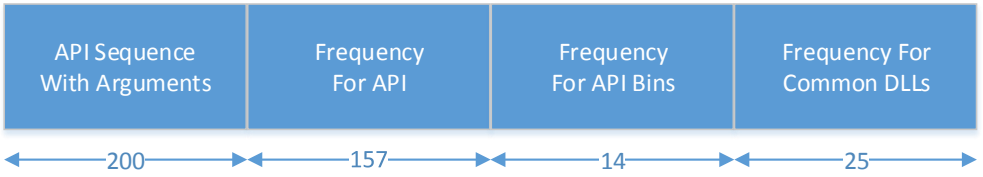


Figure 6.3: Representation 1 for Malware Detection.

Furthermore, Representation 2 consists of 596 features in total. It represents the separated sequence and again the frequency representations. An illustration can be seen below in Figure 6.4.

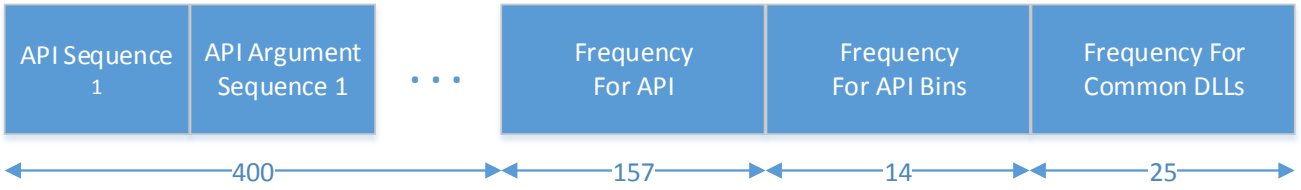


Figure 6.4: Representation 2 for Malware Detection.

Both representations are used in training and testing, thus the performance of both representations for malware detection, is evaluated.

6.4.2 Representation for Family Classification

For family classification, Representation 1 consists of 451 features in total. It uses Representation 1 for malware detection but in addition also include the signature features. Representation 1 can be seen below in Figure 6.5.

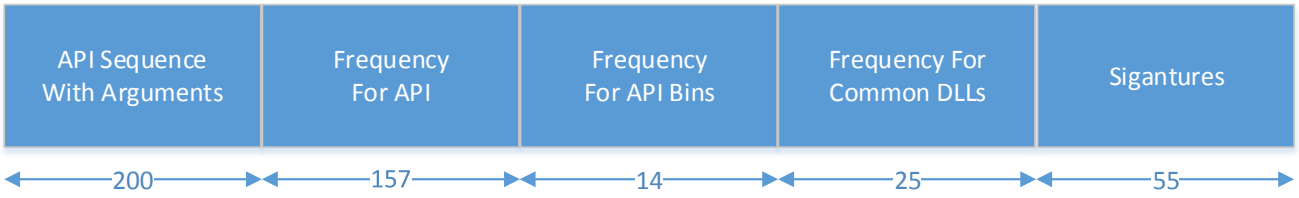


Figure 6.5: Representation 1 for Family Classification.

Furthermore, Representation 2 consists of 651 features in total. It uses Representation 2 for malware detection but will again also include the signature features. Representation 2 can be seen below in Figure 6.6.

Figure 6.6: Representation 2 for Family Classification.

Both representations are used in training and testing, thus the performance of both representations for family classification, is evaluated.

Converting Strings to Nominal Values

In Subsection 5.3.2, Random Forests classification was chosen. Because this algorithm does not directly support string features, which is the case for the sequence with input arguments, these have to be converted to nominal values. This is done using a function in WEKA called `stringToNominal`. It searches through all the features and their possible string values, and converts them to nominal. This means, that the resulting nominal feature can include many underlying labels or values, which are then weighted according to their count in the sample set. An example could be the feature `ArgNr1`, which when viewing it in WEKA can be seen in Table 6.3.

No.	Label	Count	Weight
1	0	20907	20907.0
2	RasConnectionNotificationW	44	44.0
3	NSPStartup	1	1.0
4	CreateThread	1	1.0
5	OutputDebugStringA	6	6.0
6	___CPPdebugHook	2	2.0
7	VirtualAlloc	1	1.0

Table 6.3: Strings contained in the feature `ArgNr1`.

The example in Table 6.3, shows that the feature named `ArgNr1` have 7 possible string values in the whole sample set. The high count for the string value "0" is due to how the sequence representation works. This was explained in Subsection 6.3.1. Note here, that when testing the trained model based on these features, a mapping function is used to map the nominal features in the training set with the nominal features in the testing set. Of course, the numeric features will also be mapped, but would have been unnecessary if all the features were numeric. New nominal features, that are found during the mapping, are appended to the specific feature. It does this using a the wrapper ZeroR, which is a very simple classifier that ignores all predictors in terms of probability. It bases its prediction on the majority of what it sees in the particular nominal feature in the training set compared to the testing set, see [WEKA, 2015] and [Sayad, 2015].

6.5 Chapter Summary

This chapter included an analysis of malware detection and family classification. Here, the features to be used in the study, were chosen. Afterwards, the feature extraction method from last study was compared with a new method of extraction. At last different feature representation were discussed and chosen. The main conclusions are summarized below:

1. **Analysis of Windows Parameters**
   — API calls and their input arguments were chosen.
2. **Feature Extraction**
   — Features are extracted based on API and its input arguments.
3. **Feature Representation**
   — The will be 4 different representations: 2 for malware detection and 2 for family classification.

The next chapter discusses different feature selection methods, where one is chosen for this study. Additionally, Random Forests classification algorithm will be explained, followed by a description of the evaluation metrics used for the results.

INTENTIONALLY LEFT BLANK

CHAPTER 7

CLASSIFICATION & EVALUATION

This chapter presents the different feature selection methods, after which one will be chosen. When an algorithm has been chosen, appropriate theory is presented to give the reader an idea of how it works. Furthermore, Random Forests classification algorithm is described, which was also used in the last study. At last, a description of the used evaluation metrics will be introduced. These are used for both the training and the testing phase.

7.1 Feature Selection

This section goes through the task of selecting features for the ML algorithm, that can potentially increase the predictive performance and lower generalization error. Thus, this section deals with the sub-problem:

- *How to perform automatic feature selection to filter features with high discriminative power?*

The previous study addressed Feature Selection (FS), by analyzing the features manually and testing ideas with the goal of reducing the feature space. None of the proposed methods, applied formally recognized FS algorithms or techniques, but were instead deduced by the group after analyzing the features.

This study introduce the three general FS approaches: Filters, wrappers and embedded methods along with recent work in the field of FS in general, and in relation to malware classification. The goal for this section is to find the method most suitable for this project. Before describing the methods, a brief explanation of why FS is necessary, is presented.

7.1.1 Hughes Effect

As was decided in Section 6.4, a high dimensional feature space is created in this project. Note here, that high dimensionality ranges from several hundreds to several thousands or hundred of thousands features, [Kumar, 2014]. The consequence of this, is commonly referred to as "The Curse of Dimensionality" or formally "Hughes Effect" from Gordon F. Hughes, [Hughes, 1968]. He states that the prediction accuracy, for an infinite amount of sample patterns, will converge to a global optimum as the complexity (the amount of features), $n \to \infty$. This intuitively makes sense, since the classifier will be build upon an infinite set of information, that can only yield one optimum performance as all information is known for the particular classification problem. In practice, one will have a finite set of m data samples, which will yield a different picture when increasing the complexity, i.e. the n amount of features. By interpretation by Hughes; as the complexity of the classification problem increases with a fixed amount of sample patterns, it will reach a global optimal prediction accuracy, when the amount of **statistical significant features** are reached. After the optimum is reached, and as the complexity keeps increasing, the prediction accuracy will decrease and converge to a prediction accuracy relative to the amount of classes in the classification. If the set of data samples are unequal, in relation to the classes, several optimums are

present relative to the complexity.

The potential reasons for a decrease in predictive accuracy of aforementioned, can be the presence of redundant or irrelevant features, which can lead to reduced generality and over-fitting. Here, over-fitting means, that the classifier is fitting too much to the training data and the potential noise. This leads to poor results, when the classifier is presented with data never seen before. Another consequence is the increased run-time of the applied algorithms, see [Shabtai et al., 2011]. This is the **motivation** for applying FS algorithms, that can select appropriate features: Leave out redundant or irrelevant features, that can degrade the predictive performance, and at the same time introduce relatively low computational run-time. The aforementioned assumptions will be addressed again in the training phase, see Chapter 10. The following subsections present the three common methods of FS. The definition are based on the paper [Guyon et al., 2010], which is a survey on several research papers.

7.1.2 Filters

The filter methods aim to select relevant features, i.e. remove features that does not increase the generalization significantly or not at all. Note here, that by generalization it means, how good the classifier generalizes to new data sets. Filters can narrow down the feature space without training, but are also applicable in other applications. Examples of applications along with FS are listed below:

- Pre-processing: The task of filtering "garbage data" to ensure good quality data [Wikipedia, 2015b]. Note here, that pre-processing is a general word, which can be used in many relations.
- Feature Construction: The application of using constructive operators (addition, subtractions, multiplication, division, max/min operation or average measures) to a set of features, which can then create new high-level features [Wikipedia, 2015d].
- Kernel Design: The task of finding suitable similarity functions over pairs of data points, typically applied in Support Vector Machine (SVM) [Wikipedia, 2015e].
- **Feature Selection**: This allows to reduce dimensionality of the feature space. The goal here is often to reduce the computation time for algorithms, that are often expensive computationally as Neural Networks (NN) and SVM [Guyon et al., 2010], but also to remove redundant and irrelevant features.

Usually filters use what can be termed proxy measures, not to be confused with error rate to rank feature subsets. These proxy measures can for instance be Information Gain, Pearson correlation coefficient, inter/intra class distance etc. The measures are computationally light but are still able to rank the impact or usefulness of the feature set. As a consequence, filters usually scale to high dimensional data. Even though, the computation time is relatively small compared to wrappers and embedded methods, the filters lack fine tuning towards the specific classification method. This leaves it more generalized in comparison to wrappers, which can potentially decrease the predictive performance, but at the same time reduce the risk of over-fitting, [Wikipedia, 2015c].

7.1.3 Wrappers

A wrapper considers the applied classification algorithm as a blackbox, that are then used for evaluation of the predictive performance of a feature subset. This is done by incorporating search algorithms and applying e.g. cross-validation. It can internally adjust the classification algorithms' parameters given the data, but at the same time requires no knowledge about the ML architecture or algorithm.

Usually each new subset of features, is tested with the classification algorithm and the error rates acquired after the test, are used to score the subset. Different from the filters discussed before, it uses prediction to perform the feature selection, which means that the wrapper becomes fine tuned to the specific classification algorithm used. This can potentially give higher predictive performance, in comparison with filters, but will be more sensitive to over-fitting. Additionally it suffers from being more computationally intensive as the classification has to be performed on each feature subset, [Wikipedia, 2015c].

7.1.4 Embedded Methods

Embedded methods are seen as a hybrid of filters and wrappers aiming to make use of the advantages of both methods. In contrast to wrappers, embedded methods exploits the knowledge of the ML algorithm, which are then used to make the search more efficient. Usually, the wrapper techniques only use the predictions from the learning algorithm to select the features.

As embedded methods perform feature selection as part of the learning procedure, it will usually be specific to a given learning algorithm. The pros for the embedded methods are that they are computationally less intensive than wrappers, but lack generality as they are specific to the used classification algorithm.

7.1.5 Related Work for Feature Selection

This section will go through some of the recent methods, that have been published on FS, including studies used for malware classification and detection. In the end of the section, the method applied in this study will be chosen, i.e. filter, wrapper or embedded.

The work presented in [Deng and Runger, 2012] is a tree based FS framework, that deals with penalization of the selection of new features for splitting nodes. Penalization is done, when the information gain is similar to the features used for previous splits. This means, that the method takes previous features into account when selecting them. They apply it on Random Forests and Boosted Trees, but claim that it can be easily applied to other tree models.

In [Jiang et al., 2011], the authors have developed a FS method for malware detection called Class Driven Correlation based Feature Selection (CDCBF). Features are selected based on a class driven correlation based method, that selects corresponding features for different classes of unbalanced data. Since the goal is to perform FS on an unbalanced dataset, it has combined two well known methods, namely:

- Unbalanced Data Feature Selection (DFSF), which uses the classic metric Information Gain (IG) from information theory, together with the correlation between a feature, f_1 and the classification,

e.g. positive and negative correlation [Jiang et al., 2011].

- Fast Correlation-Based Filter (FCBF), which is in the field of association analysis. Here effective features are those, which are highly correlated with the classification, but not other features. It makes use of "entropy", which is also used to calculate IG. The claim is, that if a feature, f_1 and f_2 are more correlated than the correlation to the classification, feature f_1 will be a redundant feature in relation to f_2 [Jiang et al., 2011].

The research presented in [Shabtai et al., 2011], the authors have developed a malware detection framework for android devices. Even though android devices are not in the scope of this project, it is still seen as relevant. This study chose a filter-based approach due to its fast execution time and generalization ability. Here, they emphasize, that the generalization ability is due to the filter not being bound on any classifier as is the case for wrappers. Commonly, the filter approach applies an objective function, that evaluates the features by their information content. Hereafter, an estimation of their expected contribution to the classification, is made. They used three FS methods listed in the following:

- Chi-Square (CS), which are used to test "goodness" of a fit and independence.
- Fisher Score uses Fisher's Criterion to maximize the information an observable random variable, X, tells about an unknown variable, θ, in which X depends on.
- Information Gain calculated using entropy measures.

The filter methods here return a feature ranking on each of the evaluated features.

This subsection gave a brief idea of what FS approaches can be used for classification. The next section will discuss, which method to use in this study.

7.1.6 Choice of FS Method

To decide which approach to use in this study, this section will go through the requirements for the FS method, seen by this group as were also mentioned in Subsection 7.1.1. Note here, that the following articles and encyclopedia were used: [Kumar, 2014], [Shabtai et al., 2011], [Hughes, 1968] and [Wikipedia, 2015c].

The following list summarize the requirements for the FS method. The FS method should have the ability to:

- Scale well to high dimensional data.
- Not be computational intensive.
- Rank features in terms of the best predictive performance, according to a known relevance measure (Information Gain, Gain Ratio, Pearson correlation coefficient, etc.)

It should be noted, that the performance of the wrappers and embedded methods are relative to the chosen classification method. This makes it difficult to do a fair comparison to filters, which are independent of the classification method. Nevertheless, the following will try to make a generalized comparison between the three methods.

Starting with the first requirement, it is known from aforementioned Subsection 7.1.2, that filters will generally perform best, if injected with a high dimensional feature space. This is because, both wrappers and embedded methods need to use the learning algorithm to create and evaluate the feature subspace, see Subsection 7.1.2, Subsection 7.1.3 and Subsection 7.1.4. That being said, some embedded methods are resilient towards large feature spaces, but in general, filters are seen as the top contender. This is connected with the second requirement, which states, that the method should not be computationally intensive. Here, the effects of a high dimensional feature space, could have a significant impact on the execution time. Filters are usually computationally simple and does not require to train the prediction model, like wrappers and embedded methods. It should be noted here, that there are variants of embedded methods who have minimized their execution time considerably, but it is relative to the chosen classification method. The embedded methods still require training with the classifier, which makes the performance in terms of execution time, inferior to filters. Moving to the third requirement, the FS method should be able to perform a ranking based on known ranking measures, like information gain or correlation. Here, filters generally utilize these ranking schemes, which are independent of the classifier. Wrappers still also perform ranking, but as the requirement in this study, is to use a proxy measure, as mentioned in Subsection 7.1.2, the requirement is only partly fulfilled. Some embedded methods also use information gain (RF), but as aforementioned, these are also dependent on the classification method.

In Table 7.1 a pass/fail list is made based on the requirements seen in this study. The goal here, is not to degrade one method over the other, but is entirely created based on the needs and learning goals in this project. Note here, that the table uses colors as well as symbols to indicate the level of acceptance of the requirement. Here, a **green** check mark indicates that the requirement for the given method is fully accepted. However, the **orange** check mark indicates that either some variants of the method have been found to uphold the requirement, or that the requirement is fulfilled but not specifically to the needs in this study (feature ranking). At last, the **red** indicates, that the method failed to uphold the requirement.

Requirements	**Filters**	**Wrappers**	**Embedded Methods**
Data scalability	✓	✗	✓
Min. exec. time	✓	✗	✗
Feature ranking	✓	✓	✓

Table 7.1: Pass/fail list for FS methods.

As seen in Table 7.1, filters pass all the requirements, whereas embedded methods also were a potential candidate. Looking at wrappers, the data scalability and minimal execution time is in general worse compared to filters. That being said, wrappers are very useful for feature selection in smaller dimensional feature spaces. This project aims to have a feature ranking independent of the classifier, thus leaving a lower risk of over-fitting and loss of generality.

As a conclusion to this section, **Filters have been chosen as the FS method in this study**. The next section deals with a thorough analysis of the filters available in WEKA, where an appropriate one is chosen based on preliminary tests.

7.2 Filters in WEKA

When applying FS in WEKA, two algorithms are used in conjunction, as was described in Subsection 5.3.1. This includes the algorithm that calculates the relevance measures: Correlation, Information Gain (entropy) or Gain-Ratio. The considered algorithms in WEKA, that supports these measures are the following: Correlation-based Feature Subset Selection (CFSS), Information Gain and Information Gain Ratio (IGR). Furthermore, a ranking algorithm is needed, to rank the features according to the results from the calculation. The rankers are not compatible with all measures, which needs to be taken into account, when comparing the algorithms. The rankers considered in this project will be: `Best First`, `Greedy Stepwise` and `Ranker`. The following table gives an overview of the compatibility of the algorithms in relation to rankers, see Table 7.2:

Algorithms using relevance measures	**Rankers**
CFSS	Best First, Greedy Stepwise
IG	Ranker
IGR	Ranker

Table 7.2: Overview of ranker compatibility.

The next paragraphs include a description of the considered FS algorithms:

CFSS

This algorithm is used for correlation-based FS. It works by evaluating the worth of a feature subset, where the predictive performances of each of the features, are considered together with the degree of redundancy between them, [Hall, 1998].

IG

This algorithm is used for IG-based FS. It works by evaluating the worth of a feature, by calculating the difference between information contained in the class distribution, and information contained in the class distribution given the distribution of the attribute values (features), [Guyon et al., 2006].

IGR

Like `IG` this algorithm is used for IG-based FS. In addition to calculating the IG, it calculates the ratio between the IG and the information contained in the distribution of the attribute values (features), [Guyon et al., 2006].

Only one FS and ranker algorithm will be used in Part IV. To determine which to use, a test is made based on feature Representation 1 for family classification. The results are shown in Table 7.3 (p. 53) below:

FS algorithms	Nr. of features picked	AUC	Prec.	F-Measure
CFSS+Best First	**22**	**0.972**	**0.85**	**0.835**
CFSS+Greedy Stepwise	18	0.97	0.85	0.835
IG+Ranker	451(all)	0.974	0.836	0.855
IG+Ranker	**400**	**0.974**	**0.866**	**0.858**
IG+Ranker	300	0.974	0.865	0.856
IG+Ranker	200	0.972	0.844	0.836
IG+Ranker	100	0.968	0.827	0.818
IG+Ranker	10	0.947	0.777	0.747
IGR+Ranker	451(all)	0.97	0.841	0.826
GainRatio+Ranker	**400**	**0.974**	**0.867**	**0.858**
IGR+Ranker	300	0.974	0.856	0.847
IGR+Ranker	200	0.974	0.856	0.847
IGR+Ranker	100	0.973	0.853	0.833
IGR+Ranker	10	0.792	0.793	0.52

Table 7.3: This table presents an overview of the tests made for selecting the best suitable FS-algorithm.

Only two tests were made using `CFSS`, as it was not possible in WEKA to choose how many features it should use. This option was not available using `Best First` and **Greedy Stepwise** ranking algorithms. The ranking algorithms picked 22 and 18 features for `Best First` and **Greedy Stepwise** respectively. For both `IG` and `IGR`, it was possible to use `Ranker`, which had the option to adjust the amount of features to use. These two FS algorithms are both based on IG, which is what the `Ranker` algorithm supports. With the option to adjust the number of features, additional tests were made, to find the best performance. Note here, that RF with 50 trees was used in the classification. The gray cells correspond to the best performing combination of, the amount of features, FS and ranking algorithm. The results highlighted with **green**, are the overall best performing combination. Note that the results are shown in weighted average.
Based on above results it has been **chosen to use IGR in conjunction with Ranker**. The next section describes information entropy in the context of IG and IGR. Furthermore, IG is also a metric used in the classification algorithm chosen in this study, namely Random Forests.

7.3 Information Entropy

This section introduces Information Entropy (IE) as a relevance measure for FS, which is also to split nodes in Random Forests classification.

IE is a measure of uncertainty, or unpredictability in information data. This is best illustrated by an example: Imagine that you are trying to predict the weather. This can be relatively unpredictable, as proven by forecasts that happened to be untrue. Looking at the forecast of today, whether it is sunny, rainy or overcast, can be said to give new information. In this case the IE of the forecast is large. Imagine that you look up the forecast again one minute after, the forecast shows the same result. This means

that the forecast now, is actually predictable, because of the first look-up and therefore does not contain much new information. In this case the IE is said to be low, relative to the first look-up. On the other hand, if the second forecast showed a major change compared to the first one, the IE would be larger, because new information was gained.

If one considers a binomial distribution, e.g. a coin toss, it can be assumed, that the probability of landing heads is the same as landing tails. Since the coin tosses are independent, there is no way to predict the outcome of a future coin toss using the information based on previous tosses. In this case, the IE or unpredictability, is at its maximum.

Calculating the IE in relation to the selection of features, is useful since it can remove irrelevant features. Formally the IE is calculated as the **negative of the entropy**, which can be seen in Equation 7.1, [Guyon et al., 2006].

$$H(Y) = -\sum_{i=1}^{K} \Pr(y_i) \log_2(\Pr(y_i)) \qquad [\cdot] \quad (7.1)$$

Here, $H(Y)$ denotes the information of the distribution Y, i denotes the iterator for any of the samples, K denotes the total amount of entries in the sample set and $\Pr(y_i)$ denotes the probability of sample y.

This is the general equation when calculating IE. It uses the logarithm with base two and the output is called "bits". In Section 7.2, it was chosen to use Information Gain Ratio, which includes entropy measures, that are explained now.

7.3.1 Information Gain

IG, in terms of FS, is defined as the difference in IE of the class distribution and the IE of the class distribution given the feature distribution. The feature distribution should be understood as a probability distribution of the feature set, for instance the probability of feature 1, f_1 over the whole sample set. For illustration see Equation 7.2.

$$\Pr(F) = \begin{bmatrix} \Pr(f_1) & \Pr(f_2) & \cdots & \Pr(f_M) \end{bmatrix} \qquad [\cdot] \quad (7.2)$$

Here, F denotes the feature set and M the total amount of features.

The goal is to **maximize** the IG, i.e. **minimize the unpredictability/uncertainty**. If one selects a feature, which has the same IE as the previous feature, no new information will be gained, thus the IG is low. With this understanding, the IG can be described by calculating the information, contained in the probability distribution of the classes, $H(C)$ and the information contained in the class distribution, given the probability distribution of the features, $H(C|F)$.

The definitions are shown in the following equations, [Guyon et al., 2006].

$$H(C) = -\sum_{i=1}^{K} \Pr(c_i) \log_2(\Pr(c_i)) \quad [\cdot] \quad (7.3)$$

$$H(C|F) = -\sum_{ij} \Pr(c_i,f_j) \log_2(\Pr(c_i|f_j)) \quad [\cdot] \quad (7.4)$$

Note here, that $\Pr(c_i,f_j) = \Pr(c_i|f_j)\Pr(f_j)$ where $\Pr(c_i|f_j)$ denotes the conditional information. These are identical to Equation 7.1 (p. 54), which is the general definition. The definition in Equation 7.4 is then subtracted from Equation 7.3, which is seen in Equation 7.5, [Guyon et al., 2006].

$$IG(C,F) = H(C) - H(C|F) \quad [\cdot] \quad (7.5)$$

The formulation in Equation 7.5 is the formal description of IG and is also used to split the nodes in RF. Note here, that when splitting nodes in decision trees, only a subset of the total amount of distinct features are used. How this is chosen for RF is explained in Section 7.4.

Since IGR was chosen as the relevance measure for selecting features, it is described in the following.

7.3.2 Information Gain Ratio

IGR uses the definition as in Equation 7.5, but also takes into account the distribution of features over all the samples. The feature distribution can similarly be defined as in Equation 7.3, see Equation 7.6, [Guyon et al., 2006].

$$H(F) = -\sum_{i=1}^{K} \Pr(f_i) \log_2(\Pr(f_i)) \quad [\cdot] \quad (7.6)$$

So the IGR can the be expressed as Equation 7.7.

$$IGR(C,F) = \frac{H(C) - H(C|F)}{H(F)} \quad [\cdot] \quad (7.7)$$

The division with the feature distribution, results in a ratio that expresses the IG related to how the feature values are distributed over the sample set. In [Guyon et al., 2006] they claim, that this is helpful to avoid bias, if the feature set is multi-valued with large difference between the minimum and maximum values, which is the case in this study.

The next section briefly describes RF, which is the classification algorithm used in this study.

7.4 Random Forests

RF is a decision tree based classification algorithm, that in contrast to a normal decision tree algorithm, like J48, uses an ensemble of trees to reduce variance in the classification and thereby improve the accuracy. Because the algorithm was thoroughly described already in the previous study, [Larsen et al., 2014] this

section summarizes the algorithm. The implementation of RF is based on the paper [Breiman, 2001] by Leo Breiman, who developed the algorithm. It combines multiple decision trees for each injected sample, i.e. forests, where the classification is based on a majority vote from all the trees. The trees are generated using, what is called Bagging or Bootstrapping. Bootstrapping works by generating a random subset, of the total sample set, i.e. the training data, for each tree. This means, that each tree is based on different samples when it is generated. Because of this, the generated trees are assumed to be noisy and unbiased to each other, resulting in high variance and decorrelation among each other. The algorithm can be expressed by the following steps:

1. Create T amount of trees and for each tree do following:
   (a) Generate a bootstrap sample $\mathbf{Z}^*$ of size N from the whole sample space (training data).
   (b) Grow the tree T_i based on recursively repeating the following at each node:
      i. Randomly select m features from a feature subset, of the bootstrap sample, as candidates for splitting the data.
      ii. Pick the best feature split among the m random features based on highest IG.
      iii. Create two daughter nodes.
2. Output ensemble of trees (forest).

Here, $m = \mathbf{int}\log_2(M) + 1)$ where M is the maximum number of input features. Since the features are picked randomly, it will create high variance among the trees, which is then lowered when using the majority vote, or average of all the trees. Research on RF is based on [Breiman, 2001], [de Freitas, 2013] and [Hastie et al., 2009] and this brief summary, is based on the previous work, see [Larsen et al., 2014].

7.5 Evaluation of the Classifiers

This section presents the evaluation metrics, used when evaluating the detection and classification. Because, theory about these metrics have already been thoroughly described in the previous report, this section will only briefly describe them.

7.5.1 Confusion Matrix

The confusion matrix provides an overview of the predictions made by the classifier. This includes the following four metrics:

- True Positive (TP) is when a condition is correctly predicted as positive.
- True Negative (TN) is when a condition is correctly predicted as negative.
- False Positive (FP) is when a condition is incorrectly predicted as positive.
- False Negative (FN) is when a condition is incorrectly predicted as negative.

These definitions are directly defined by the confusion matrix. An example of a binary case can be seen in Table 7.4 (p. 57). [Wikipedia, 2014a] [Manning, 2014] [Larsen et al., 2014]

		Condition	
		Positive	Negative
Predicted	Positive	**TP**	**FP**
	Negative	**FN**	**TN**

Table 7.4: Confusion matrix for a binary classifier.

A confusion matrix will be used for evaluation in Chapter 11.

7.5.2 Additional Metrics

Like the last study, several metrics will be used to evaluate the classifier. This includes: TPR, FPR, Positive Predictive Value (PPV) and F-Measure (F-M). The definitions are described in the following equations:

$$TPR = \frac{TP}{TP + FN} \tag{7.8}$$

$$PPV = \frac{TP}{TP + FP} \tag{7.9}$$

$$FPR = \frac{FP}{TN + FP} \tag{7.10}$$

$$\text{F-M} = 2 \cdot \frac{PPV \cdot TPR}{PPV + TPR} \tag{7.11}$$

These metrics are used in both Chapter 10 and Chapter 11. [Ng, 2014] [Wikipedia, 2014d] [Manning, 2014] [Konstan, 2014] [Wikipedia, 2014c] [Wikipedia, 2014b]

ROC Curve

Additionally, the testing phase will include a Receiver Operating Characteristic (ROC) curve to describe the final overall performance of the individual classes. The ROC curve is defined by the TPR and FPR distributions, where a threshold is varied to depict the performance at different TPR and FPR levels. This is defined as the following:

$$TPR(T) = \int_T^\infty P_1(T)dT \tag{7.12}$$

$$FPR(T) = \int_T^\infty P_2(T)dT \tag{7.13}$$

Using the defined curves, one can calculate the AUC, which is a metric that defines how the classifier overall performed. The steeper the curve, the larger the AUC, since it is defined by the integral of the ROC curve. [Wikipedia, 2014c] [Konstan, 2014] [Gönen, 2007] [Krzanowski and Hand, 2009]

7.6 Chapter Summary

This chapter, described the FS algorithm that will be used in this study, along with appropriate theory about the used relevance measures. Furthermore, RF algorithm was explained, followed by a description of the evaluation metrics, used in the training and testing phase. Below a list summarizes the main conclusion of this chapter.

1. **Feature Selection**
   — Here it was chosen to use IGR relevance measure, along with Ranker for selecting features in the training phase.
2. **Information Entropy**
   — Relevant theory about Information entropy, including IG and IGR were described.
3. **Random Forests**
   — Theory about RF were described.
4. **Evaluation Metrics**
   — The chosen evaluation metrics were: TPR, PPV, F-measure and AUC.

Then next chapter includes the system requirements for this project.

CHAPTER 8

SYSTEM REQUIREMENTS

This section presents the requirements for the system. When developing the different software modules, they have to comply with the requirements. It is chosen to set up requirements for each of the modules, where the main focus is on I/O relationships. No requirements are set for the predictive performance of detection and classification. Though, the measures for evaluating the classifiers are presented.

The testing includes a Training phase, where the FS algorithm and data set are tweaked with the goal of enhancing the performance of the classifier. Afterwards the Testing phase, where the actual evaluation of the predictive performance are discussed. The requirements listed next, are tested while developing each of the modules, hence the system works as expected when discussing the results in Part IV.

Data Generation:
Cuckoo should be able to control the VM hosts and connect with both InetSim and MongoDB (off-the-shelf tools).

Label Extraction:
This module, should correctly extract the labels for family classification.

Feature Extraction:
This module, should correctly extract the wanted features in detection & family classification.

Feature Formation:
This module, should correctly format and split the features into a file, compatible with WEKA.

WEKA - IGR/Ranker:
This module, should be able to run the IGR with Ranker (off-the-shelf tool).

WEKA - Random Forest:
This module, should be able to run RF (off-the-shelf tool).

Note here, that the Data Generation module is identical to last study, but Cuckoo has been updated to the latest version. This module is tested as a whole, which means that Cuckoo, InetSim and MongoDB should all work together.

As aforementioned, no requirements are listed for the performance of the system. It is evaluated based on the evaluation measures described in Section 7.5, where the findings will be thoroughly discussed.
The evaluation metrics are the following: **TPR**, **PPV**, **F-measure** and **AUC**. These measures are used in the evaluation, and the requirements are used during the development and debugging of the system, i.e. Design and Implementation, described next.

INTENTIONALLY LEFT BLANK

Part III

Design & Implementation

INTENTIONALLY LEFT BLANK

CHAPTER 9

DESIGN & IMPLEMENTATION

This chapter presents the design and implementation of the modules used in this study. The description of the modules are build up by first presenting a flowchart, after which only relevant code snippets are shown. The complete code can be found on the the CD in the Code folder. The next section presents an overview of the system.

9.1 System Overview

This section presents an overview of the Malware Detection & Classification Framework. The system is depicted in Figure 9.1, where the structure has been slightly changed compared to the last study. The system consists of three parts, namely: Data Generation, Data Extraction and Detection & Classification.

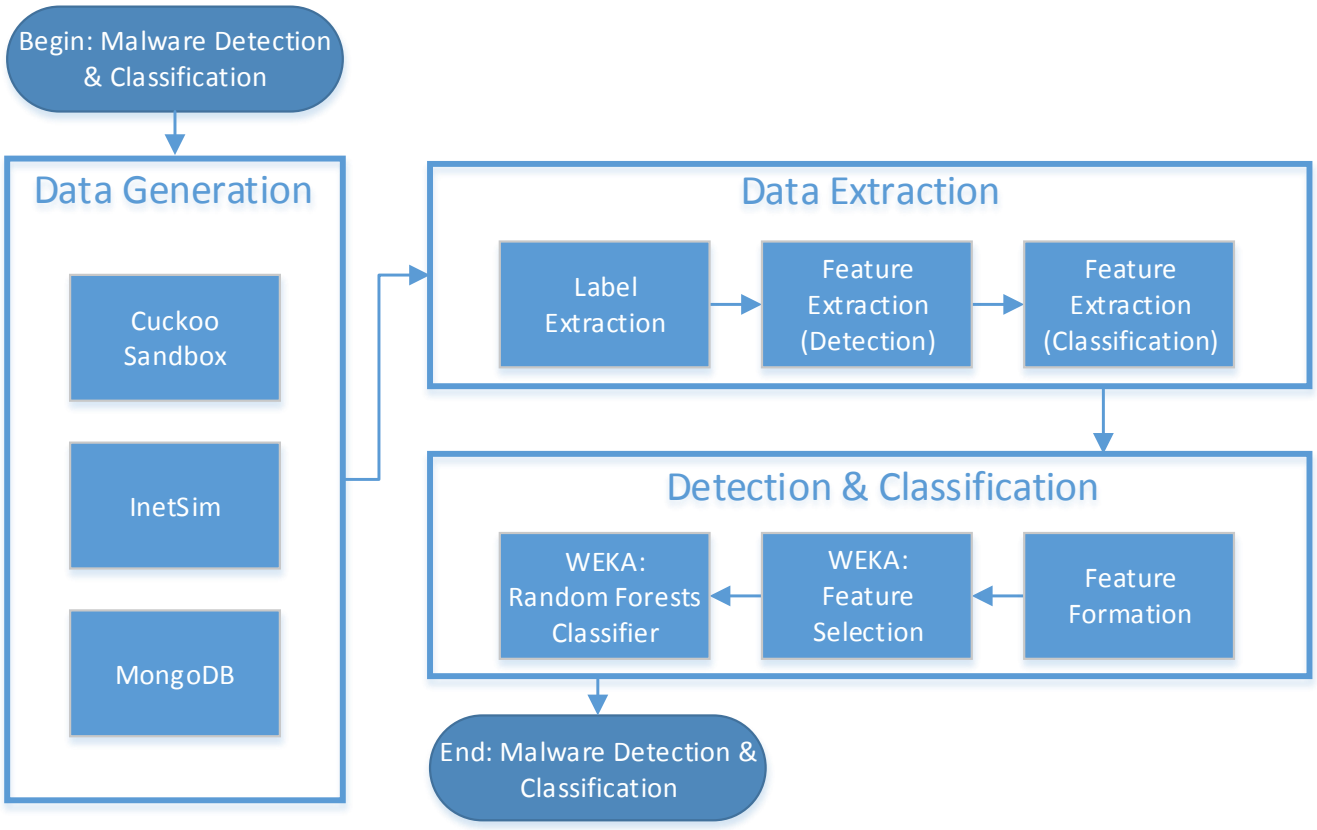


Figure 9.1: Overall flowchart of the Malware Detection & Classification framework.

Since most of the Data Generation was developed in the last study, the design and implementation of this part will only be summarized in the next section.

Data Generation: The modules included in this part, are responsible for running the dynamic analysis and storing the reports. It consists of Cuckoo Sandbox, which controls the malware execution. For the

sake of having an as realistic environment for the injected malware as possible, InetSim emulates Internet services. Lastly, all recorded information from the experiments are saved in MongoDB, a NoSQL database.

Because no changes were made in this module, besides updating Cuckoo and MongoDB to the latest versions, it is not described again in this report. Note, that all requirements in Chapter 8 are met. For implementation of this, see [Larsen et al., 2014], which is located on the CD in the folder `Previous Study\`.

Data Extraction: This part consists of three modules responsible for collecting all the needed data from the database. This includes the label extraction, where the labels for the classes are retrieved. Afterwards feature extraction is performed for both detection and family classification. It should be noted here, that the classification uses the same features as detection, but as an addition it includes signatures from the malware families.

Because changes were made in all the functions for this module, they will all my described in this chapter.

Detection & Classification: Because, this study uses WEKA ML toolbox, the first part of this module is responsible for converting the concatenated features into ARFF. When the formation is done, two files are ready for pre-processing. These consists of the testing and training set. The pre-processing module, WEKA, here utilizes FS based on IGR, where ranker is used to rank the features accordingly. When the FS is done, the next module is responsible for performing both the detection and family classification. Note here, that these modules are seen as a blackbox in WEKA, since these are not designed and implemented by this group.

Because the WEKA modules are seen as a blackbox, only Feature Formation will be described in this chapter. Note, that both requirements in Chapter 8, for the WEKA modules are met. Again, a description of this can be seen in [Larsen et al., 2014], which is located on the CD in the folder `Previous Study\`.

9.2 Label Extraction

This section describes how labels are extracted from the database. As mentioned in Subsection 5.5.1, families are used as labels for the classification. Note, that the label extraction is similar to the program used last semester ([Larsen et al., 2014]), but with slight modifications.

The requirements for this module can be found in Chapter 8. It was decided earlier in this report in Section 3.2, that this study use family labels provided from Microsoft. This is because Microsoft supports the CARO standard, which makes it more straightforward to extract the labels.

Viewing the flowchart in Figure 9.2 (p. 65) the collection is first created on the database. Using the cursor when querying the created collection, a file is created where the information is stored. When the information have been saved a function extracts the needed labels together with the filename for each malware sample. The filename for the malware sample is needed to keep track on the each individual

sample, when the feature representations are generated. These modules are described later in Section 9.3. The extraction is saved to a new file, after which another function now sorts and counts the labels. This is used to create statistics about the distribution of the malware samples in the database. It is also saved to a file, where the results for instance, were used to create the bar-plot in Figure 5.6 (p. 29). It should be noted, that the reason for creating these files, are to store the information so the program only has to run once. It is also easier to load the files in the other modules, which can then run individually when testing the functionalities.

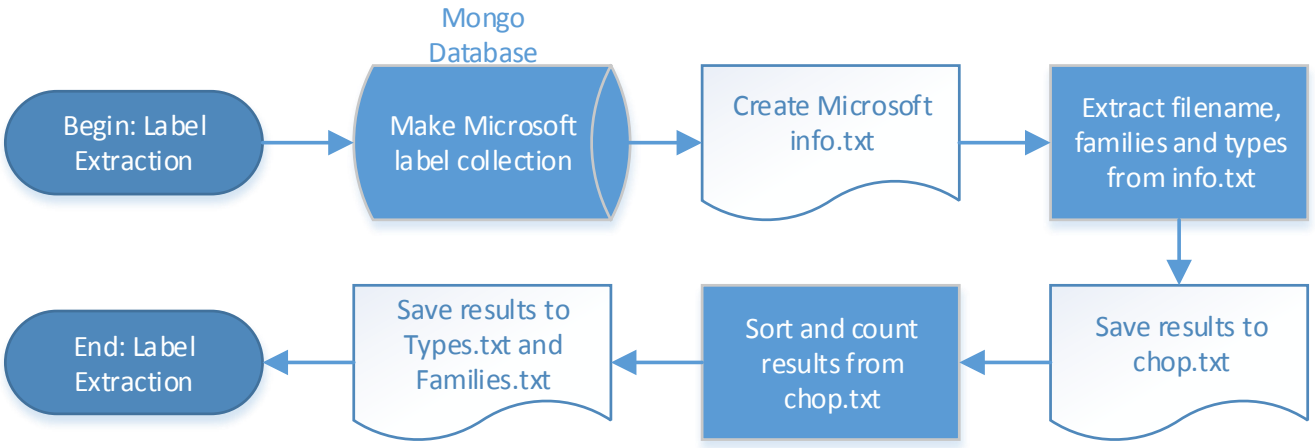


Figure 9.2: Flowchart of the Label Extraction program.

To give the reader an idea of how each of the labels along with the filename are extracted and stored, the code in Code Snippet 9.1 is shown. This code is slightly different from last semester because Microsoft was not used. As can be seen in Line 3 and Line 6, two splits are made. This is because each line in `info.txt`, which is the file created after the collection has been made, consists of two strings separated by a space. Therefore, the first split separates these two strings using the regular expression `'\s'`.

Code snippet 9.1: How labels and filename are extracted from Microsoft's AV-program.

```
1  for numLine in dbInput:
2      # Because filename have alot of ./!:, we split that first by space
3      nameOfFile = re.split('\s',numLine)
4      fileName = nameOfFile[0]
5      # Split if space (\s), '.', ':', '/' and '!'
6      chopIt = re.split('\s|[:,/,.,!]',nameOfFile[1])
7      # Save family labels
8      family = chopIt[2]
9      # Output to file
10     output.write(fileName + ' ' + typeOfMalware + ' ' + family + ' \n')
```

The second split, uses the second entry of `nameOfFile` and separates the strings by the symbols known from the CARO standard, see Subsection 3.3.1. The strings with family are saved to a file. This code runs for all the samples in the created collection. The sorting and counting are identical to the code from last semester and will therefore not be described here. The requirements for this module, seen in Chapter 8 are all met. The code for this module can be found on the CD in the folder `Code\LabelExtraction\`.

This concludes the Label Extraction, which described how Microsoft labels were extracted from the database. The data output from this module will be used when creating the feature representations in the Feature Extraction module described in the next section.

9.3 Feature Extraction

The Feature Extraction module is slightly different from the last study, and now also depends on if malware detection or family classification are used. A new collection, which includes all APIs and corresponding arguments for all samples, has been created. Below in Figure 9.3, the process of extracting and generating the representation is seen.

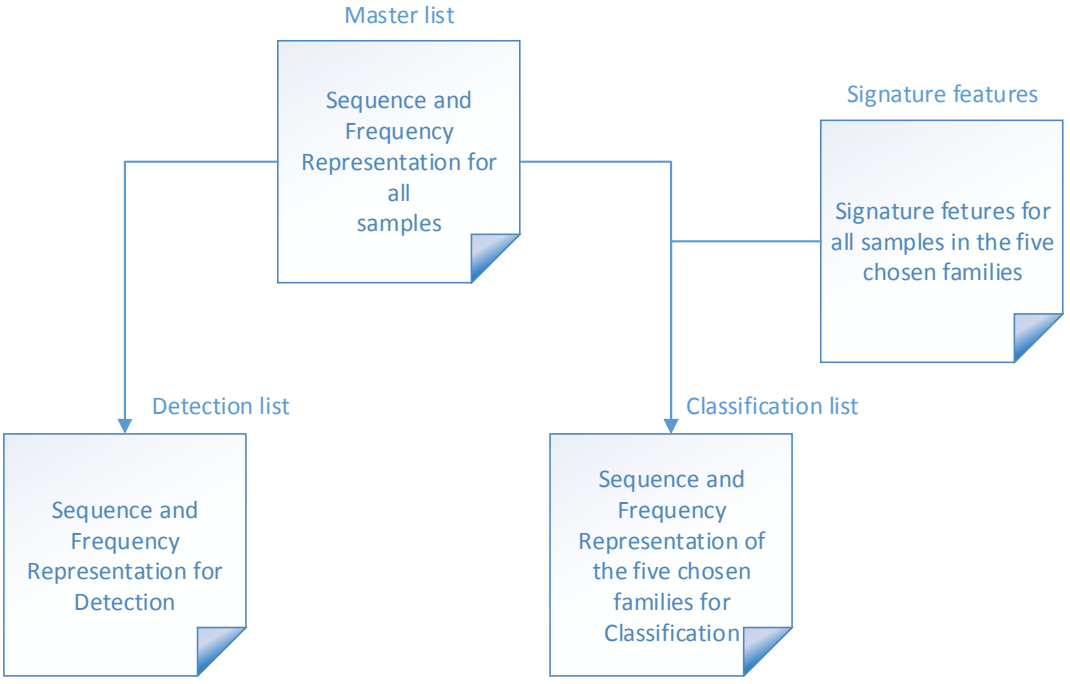


Figure 9.3: How the analysis data is extracted and used.

This figure describes the foundation of how the program should be constructed. To ensure less computation when working with Big Data, it is decided to extract the sequence and frequency features for all samples first. These features are then stored in a pickle file according to the sequence and frequency representations. Since family classification needs signatures as well, a separate list is created for those. After all the extractions are finished, another process will generate the final representation for both detection and family classification. Here, the needed data is extracted from the Master list and for classification, also the signature features. The reasoning behind actually extracting information from the Master list, and not just extract it from the database, is to save time. This is because the Master list is only one list and contains all the information, that can potentially be used. The database includes many nested lists and not all information is needed. Using nested loops, increases the computation time significantly and because it is needed to iterate through the list several times, this is the most efficient approach. In Figure 9.4 (p. 67), the flowchart of the whole process for extracting and generating the representations can be seen.

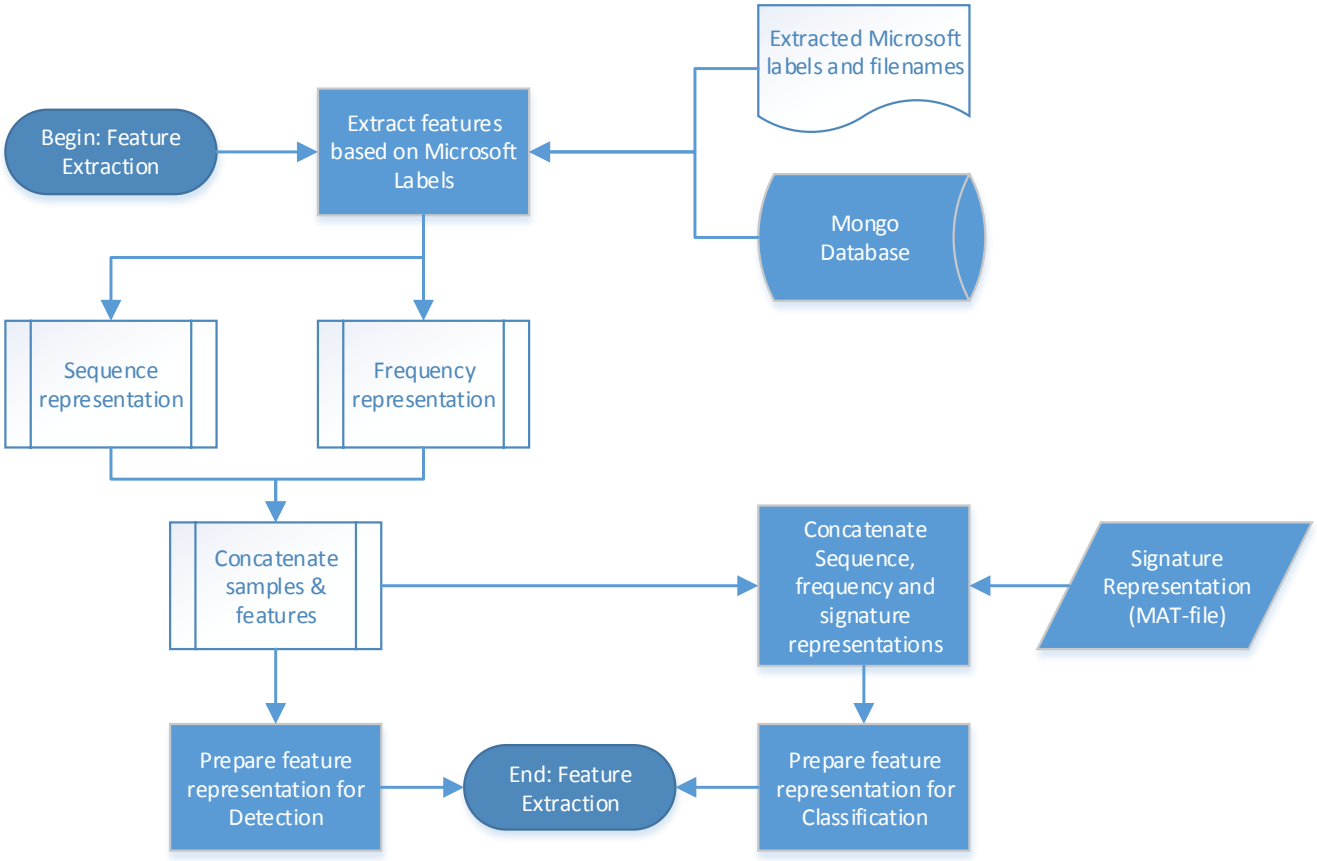


Figure 9.4: Overall flowchart of the Feature Extraction module.

First the list of all Microsoft labels and filenames are loaded, such that each sample detected by Microsoft can receive its type and family label. Features for all samples are extracted, but samples without a label from Microsoft, will be labeled unknown. For each sample, sequence and frequency features are extracted and represented as described in Section 6.4. The extraction of the features happen in the same program, but is documented individually. Afterwards, the representations for malware detection and family classification are generated. The representation for family classification is first made, after the signature features have been extracted, as these will be concatenated with the specific family samples in the Master list. The families chosen in this study were listed and described in Subsection 5.5.1. Further, all requirements in Chapter 8 are met for this module.

Throughout this section, the implementation of sequence, frequency and signature extraction is documented. At the end of this section, the final preparation for the representations is described. The code used for extracting both the sequence and frequency representation, can be found for both malware (ExtractRepresentationMalware.py) and cleanware (ExtractRepresentationCleanware.py) on the CD in the folder `Code\ExtractDataFromMongo\`.

9.3.1 Sequence Representation

The features chosen in Subsection 6.3.1, need to be extracted and represented as described. The extraction is done on a per sample basis, by iterating through each API, and extracting the API name

including its potential input argument for "FunctionName". Before running this program, a script is executed, which extracts all input arguments for all APIs. This script is located on the CD in the folder `...\Code\ExtractDataFromMongo\ExtractAnalysisDataForCleanware\extractArgbyFunc.py` and `...\Code\ExtractDataFromMongo\ExtractAnalysisDataForCleanware\extractArgbyFunc.py`. The script generates two .MAT files, which need to be loaded by this extraction program in order to assign an ID for the 3,500 most used malware arguments and 2,000 most used cleanware arguments. The assigned ID, is needed only for Representation 1. Furthermore, each API needs to be assigned with an ID number, which is done using a collection from the database, that includes all the different APIs. From extraction, one list containing 157 APIs is created. This list can then be used to compare the API name and use the index as a unique ID number.

The code for extracting the API sequence, is found below in Code snippet 9.2, which is executed for each PID of the sample. For every API from the PID, the ID number or API-name, is stored in its respective place in the sequence. Before storing the API, it is ensured by the `if`-statement, that no more than 200 APIs are stored, which is the value of `seqLength`. Also, `j+control`, ensures that the placement of the API in the sequence, is correct. `Control` is used to save the order number of the APIs from the previous PIDs. When extracting the API ID number or name, `[i][j]` is used, since the collection the data are extracted from, is structured as nested lists. This means, that each PID will include a list with APIs, whereas each API will include a list of arguments etc..

Code snippet 9.2: How API sequence features are created for each of the samples.

```
1
2 for j in range(0, nrOfAPIs): # Save each API in sequence with its unique ID from apiList
3   if j + control  <= (seqLength-1): # Only want the first amount of seqLength. Use control
        variable to begin where it left from last PID, if any.
4     if Rep == 1:
5       seqAPIList[j + control] = apiList.index(entry['api'][i][j]) # Assign a unique API to
            the sequence list
6     elif Rep == 2:
7       seqAPIList[j + control] = str(entry['api'][i][j]) # Assign a unique API to the
            sequence list
```

It should be noted, that `Rep` denotes if the feature is for Representation 1 or 2. The extraction of the API's input argument, can be seen below in Code snippet 9.3. Here, the argument is only extracted if the argument name is `FunctionName`, which is specified in the variable `APINameToUse`. If the name is found, the argument is extracted, as long as the limit of the extracted APIs has not been met. For Representation 1, the feature is extracted by a `try`, using the index function to extract the ID of the argument. This is needed, because if the argument is not in the assembled list, the index function will fail. To ensure, that the program continues if the argument does not exist in the list, a zero is inserted instead as an `exception`. For Representation 2, the name of the argument is inserted, and can include all arguments for the APIs.

Code snippet 9.3: How API input argument sequence features are created for each of the samples.

```
1 if entry['name'][i][j][k] == APINameToUse[0]: # Used for API arguments
2   if j + control <= (seqLength-1)
```

```
    if Rep == 1:
      try:        # Try since the name might not exist
        seqAPIArgList[j + control] = indexForVal.index(entry['value'][i][j][k])
      except ValueError: # If not exist, only except if returns that error
        seqAPIArgList[j + control] = 0 # Add 0 to list, to ensure that index will be reset
    elif Rep == 2:
      seqAPIArgList[j + control] = str(entry['value'][i][j][k])
```

The sequence for Representation 1, is generated by combining each entry in `seqAPIList` and `seqAPIArgList` by a -. Note here, that the full list includes 200 features. Furthermore, the sequence for Representation 2, needs to be generated using a list, where the API and argument sequence alternates, thus API1, ARG1, API2, ARG2,..., etc.. The complete list includes 400 features.

9.3.2 Frequency Representation

The same variables and lists used before also exist for this function, since they are extracted during the same iteration. The features were described in Subsection 6.3.2. Below in Code snippet 9.4 the extraction of both the frequency of APIs and API bins is found. It should be noted, that the 14 different categories from Cuckoo, are denoted as API bins in this study. These are made as a list in the beginning of the program, such that each bin has a unique ID number related to its index. The frequency for each feature is made by adding 1 to the entry of the index for the API or API bin.

Code snippet 9.4: How API and Bin frequency features are created for each of the samples.

```
FreqBinList[APICategory.index(entry['category'][i][j])] += 1
FreqAPIList[apiList.index(entry['api'][i][j])] += 1
```

Below, in Code snippet 9.5, the extraction of frequency for different DLLs is seen. Note, that the different DLLs used, have been made as a list in the beginning of the program. The different DLLs were picked by running another script, from which two .MAT files are created. These files are used to decide, which DLLs to use for extraction, since they show the overall frequency for each DLL file, in the whole sample set. The script to create these two files, can be found on the CD in folder `Code\ExtractDataFromMongo\DLLextract\extractDLLNames.py`.

From the code in Code snippet 9.5, it is seen, that if the argument contains the name `FileName`, which is stored in the variable `APINameToUse`, it compares the value of the argument. Before comparison, both variables are converted to lower case string letters. Again, as for frequency bins, 1 is added to the entry, which is equal to the index of the DLL, found in the DLL list.

Code snippet 9.5: How the DLL frequency features are created for each of the samples.

```
if entry['name'][i][j][k] == APINameToUse[1]: # Used for DLL frequency
  for item in DLLsToLookFor: # Loop through all index of DLL
    if item.lower() in entry['value'][i][j][k].lower(): # Convert both the extracted info
        and the name to compare with to lower case
      DLLList[DLLsToLookFor.index(item)] += 1 # If match add one to the index of the DLL
```

Representation of frequency features are made by concatenating all the features extracted for each sample. The order will be the frequency of APIs, API bins and at last the specified DLLs.

9.3.3 Signature Representation

This subsection explains the design and implementation of extracting the signatures and representing them as features. The idea of using signatures as features was explained in Subsection 6.3.3 and the complete list can be found in Appendix A (p. 125). To summarize, the idea was to find signatures specifically for each family. The assumption was, that because families mostly are named based on their source code, they must have some common characteristics. This could for instance be how they name files, mutexes and REG keys, or which they will access. A good example is the Hupigon family, which often uses the string `GrayPigeon`.

The function, which is responsible for collecting the signature features are then dependent on the signatures found, by manually searching on the Internet. If more time had been available, more and probably better signatures, could have been found. Additionally, it is imagined that the signatures can be extracted from a private AV-vendor's database, containing the behavior summaries. The flow of the function is seen in Figure 9.5.

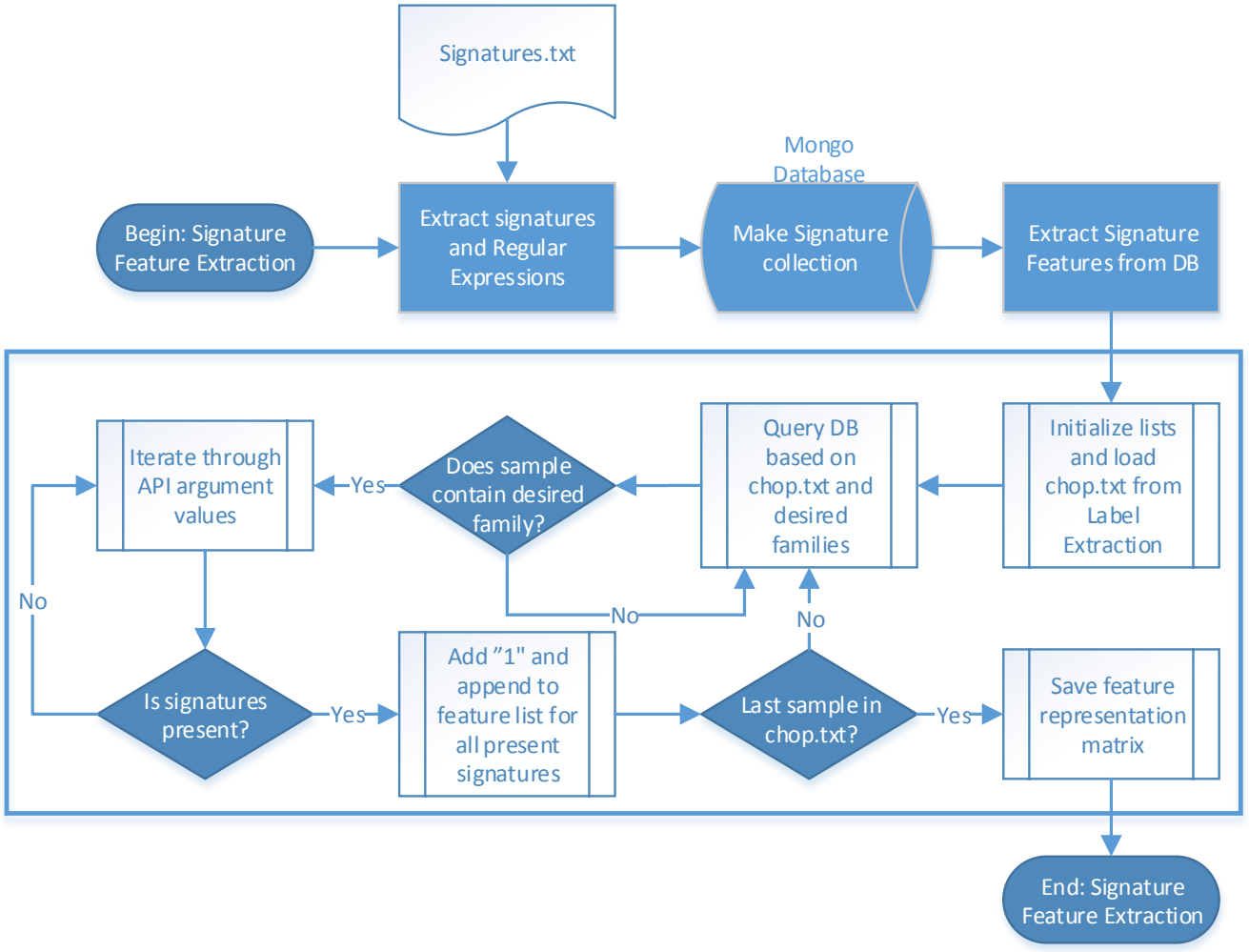


Figure 9.5: Flowchart of extraction of signature features.

In the beginning of the function, the signatures are first extracted from `Signatures.txt`, which contains

the signatures. A function then converts these signatures into Regular Expression (RegEx), which are used to query the database. Using all the signatures as a RegEx, they are matched with the API argument values from the malware samples. After the matching, everything is grouped and saved into a new collection. This collection contains all the information needed for extracting the features, which happens in the next function. As depicted in the flowchart, the function "Extract Signature Features from DB" points to a large rectangle, which denotes the sub-processes inside. Firstly, all the required lists are initialized and the `chop.txt` from the Label Extraction module, is loaded. Based on the malware samples and the five family labels, a `for`-loop starts, that iterates through all the samples in the "chop"-list. The chop-list contains the filenames for the five families along with their labels. If there is a match with either of the five families, the loop continues, and if not, the loop moves on to the next sample. If there is a match with the family, the filename is used to query the database. Here the API argument values, that contain either of the signatures, are retrieved. Now the function iterates through the API argument values and match all the signatures. If a signature is present, binary "1" is added to the feature list, that corresponds to this specific sample. These iterations continue until the function reaches the last sample in the "chop"-list. Afterwards, a feature representation matrix is saved, which contains the feature lists for all the family samples. This is saved in a MAT-file, which is used again when the feature representations are concatenated in Subsection 9.3.5.

The code responsible for iterating through the samples and the API argument values, is seen in Code Snippet 9.6. A query is made to the `SigFeatures` collection in Line 2, where `virusName` is the filename. This means, that a cursor is retrieved, containing the information for the specific sample. Because, the cursor contains nested iterable objects, it is possible to first loop through the outer object. Then the object containing the API argument values are saved and a new loop starts, which iterates through all the objects in `argval`.

Code snippet 9.6: How features are created for each of the family samples.

```
1  # Get cursor from filename query
2  SigColCursor = db.SigFeatures.find({"filename": virusName}, {"argval": 1}).limit(1)
3  | #NB: Code have been removed between line 2 and 4 for readability
4  for argvalCursor in SigColCursor:   # Iterate through the cursor
5    x = argvalCursor['argval']  # Save argval from the collection
6    for u in range(0, len(x)):  # Iterate through all instances of argval
7  # Iterate through the number of signatures in the SigList
8      for entry in range(0, len(SigListRE)):
9  # For every signature search in all instances of argval with regular expression of the
       signature
10       if re.search(re.compile(SigListRE[entry], re.IGNORECASE), x[u]['value'], flags=0):
11         seen[entry] = 1 # Add 1 to each entry of the feature list for each sample
```

In Line 8, the third and last loop starts. This loop iterates through the list of signatures, which were converted to regular expressions, to ensure that each argument value is matched with each signature. This is the same concept, as when the signature collection was created. The function `re.search` is used to match the signature with the argument value in Line 10. Note here, that `re` is a library which contain a lot of RegEx operations. If there is a match, binary "1" is added to the feature list, here called `seen`, for the corresponding malware sample. Afterwards, `seen` is appended to another list, which is the rep-

resentation matrix. In Python, appending a list to another list will still be a list, but instead of calling it a matrix they formally call it nested lists. When the function has reached the last sample, the feature representation is saved to a MAT-file, as was also explained in the flowchart.

The code for this module can be found on the CD in the folder `Code\MultiClassClassification\`. The next section describes how the representations for detection and family classification are made.

9.3.4 Representation for Malware Detection

The representation for malware detection was described in Section 6.4, where it is noticed that it is similar to what is being extracted from the Master list. The difference is the composition of cleanware and malware, where multiple sample sets are created. For training, the following sample sets are needed: All cleanware together with 1k, 5k, 10k, 20k and all malware samples. It should be noted, that only Microsoft detected malware samples are used. This is done by creating 5 text files with the name of the malware samples, that should be used in combination with the cleanware samples. These files are used to create the sample sets by extracting the representations for the samples found in the text files. The names of the malware samples are found by the number of samples, that are present for each malware family. The script used to create the text files can be found on the CD in `Code\MalwareDetection\FindMalwareNamesForDetection.py`. The script generates a list of all the families, and then order them with largest first. For the text files including the names of 1k and 5k malware samples, one sample from each of the largest families, is inserted in the file. Since there are approximately 6,500 different families, the list for 10k and 20k, contains multiple samples from the same family. The list of families are iterated through multiple times, and for each iteration, a sample is inserted to the text file and removed from the family list, such that duplicates are avoided. The list of families are shrinking, since some families only contain one sample. This means, that families with most samples, are included more frequently in the two lists. Also, a text file with all samples are created, which are used in the program below.

In Figure 9.6 (p. 73) the representation of cleanware and specific malware samples are concatenated. First, the representation for all samples are loaded, together with the list of malware samples, that needs to be used. Afterwards all Cleanware samples are appended to their respective lists, in which all the labels for the cleanware are relabeled to "cleanware". At last, all selected malware samples are appended to the same list as cleanware and relabeled to "malware". When the list is completed, it is saved, such that it can be converted to an ARFF file in the Feature Formation module, described in Section 9.4.

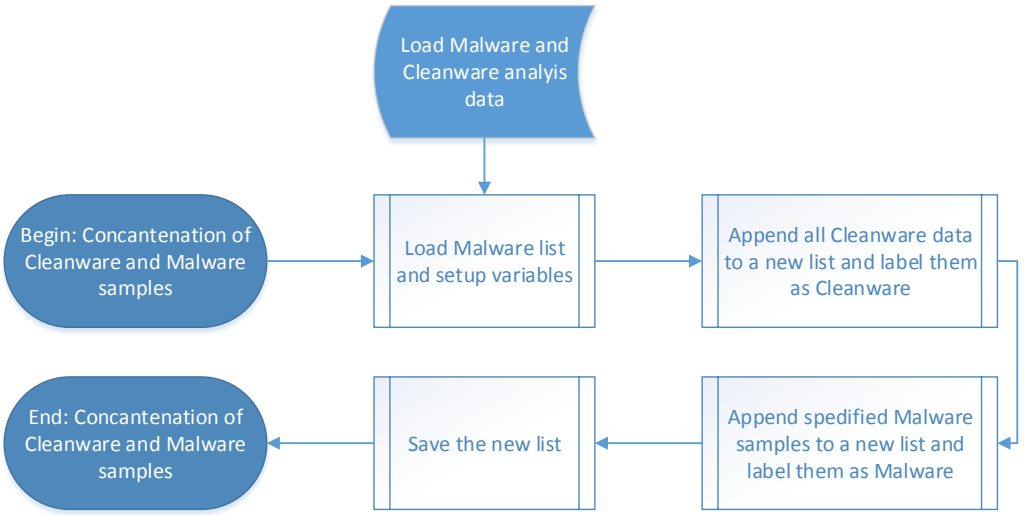


Figure 9.6: Flowchart for concatenation of cleanware and malware samples.

The code for searching through the list of samples to find the specific malware, are found in Code snippet 9.7. To optimize the search, each sample name is compared with each name from the text file, until a match is found. When a match is found, all data is copied to multiple lists, one for family label, filename and data. As seen, the family is relabeled as malware. This is done, since detection is a binary classification, namely cleanware and malware. Afterwards, the for loop breaks, since the name was found.

Code snippet 9.7: How malware samples are extracted.

```
for i in range(0,len(filename1)):   # Take every filename from the list
  for j in range(0,len(MalList)): # Seach in the master list for the filename
    if filename1[i] == MalList[j]:  # Check for match
      NewFamilies.append('Malware') # Relabel family to malware
      NewFilename.append(filename1[i]) # Append the filename to new list
      NewData.append(data1[i])  # Append the data to new list
      break # Do not search for this filename anymore
```

This concludes the creation of the representations, that are needed for Malware Detection. It should be noticed, that this code needs to be executed for both representations, for all the sample sets. The code for concatenation can be found on the CD in the folder
`Code\MalwareDetection\mainToProduceDetectARFF.py`.

9.3.5 Representation for Family Classification

Because the feature representation is different for the family classification, a slightly different representation is made. In Subsection 9.3.2, Subsection 9.3.1 and Subsection 9.3.3 it was explained how the three main representations were designed. This subsection explains how the combined frequency and sequence representations, are combined with the signature representation. This combination is made to secure better discrimination when dealing with multiple classes. The flow of the program is depicted in Figure 9.7 (p. 74).

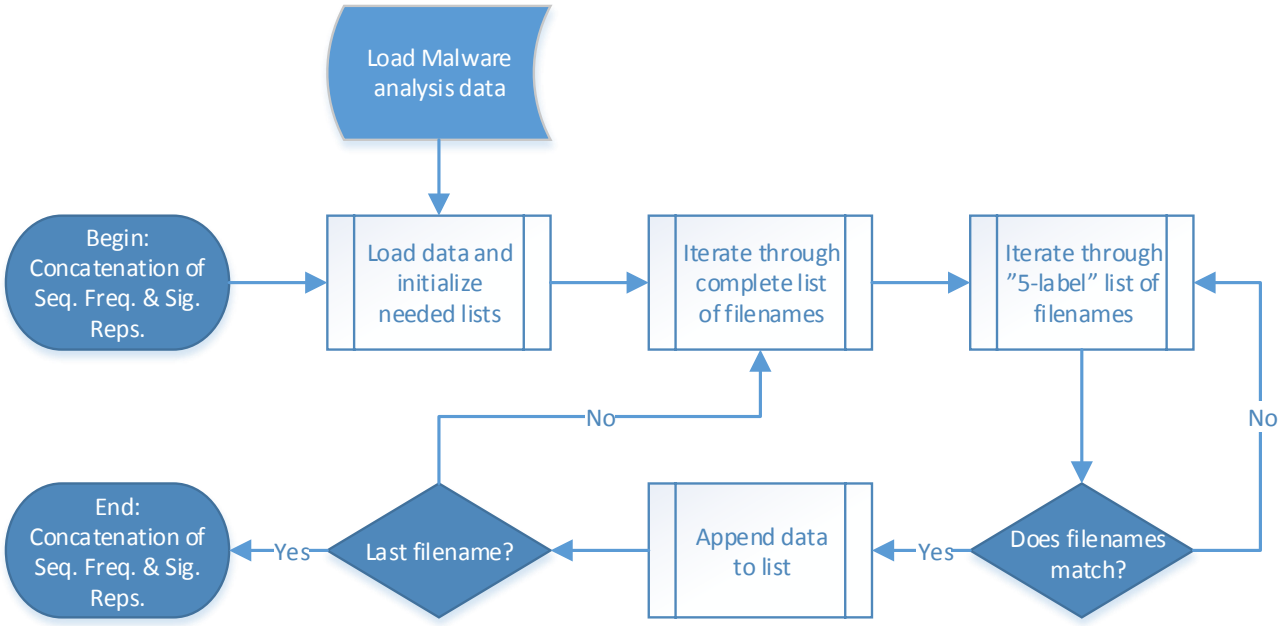


Figure 9.7: Concatenation of features.

Firstly, the data from the detection and signature representation are loaded. Initialization of the lists are done, which are needed when creating the feature representation. The program afterwards consist of two `for`-loops, in which the first one iterates through the list of all family samples. The second loop iterates through the list of the five chosen families. At each iteration of the two loops, the corresponding filenames of the two data lists, are compared. The first loop keeps track on the filename in the full sample list, whereas the second loop keeps track on the filename of the sample list from the five families. If there is a match between the filenames, the features from detection are concatenated with the signature features. Furthermore, the family name and filename are also saved for backup. After the matching is complete, the lists are saved in a dictionary, which are then stored in a pickle-file. This data format is a common mechanism for data object serialization, and is commonly used in Python. By serialization it means, that the Python object hierarchy is translated into a byte stream for storage, and can thereby by reconstructed later using the reverse operation. The formal term in Python programming is ”pickling” whereas the reverse operation is called ”unpickling”, see [Python Documentation, 2015].

To give the reader an idea of how the program is implemented see Code Snippet 9.8. As mentioned earlier, the program consists of two `for`-loops, where filenames of the two data lists are compared by a simple `if`-statement.

Code snippet 9.8: Concatenation between signature rep. and detection rep..

```
for i in range(0,len(filename2)): # Iterate through complete family list.
    # Iterate through 5 family list which include signatures
    for j in range(0,len(filename1)):
        if filename2[i] == filename1[j]:  # If match between filenames, concatenate
            # Saving all needed information
            NewFamilies.append(familyName1[j])
            NewFilename.append(filename1[j])
            NewType.append(typeName1[j])
```

```
9            NewDataVec = data2[i]+data1[j]
10           NewData.append(NewDataVec)
11           break # Start with next filename in the list
```

When there is a match, the needed information are appended to the appropriate lists. Notice the `break`-statement on Line 11, which is for flow-control. When there is a match, there is no reason to continue searching for more matches, because the two lists only contain unique filenames. This means, that only one match for every entry in `filename2`, is possible. The code for this program can be found on the CD in the folder `Code\MultiClassClassification\`. The next section explains how the feature representations are formatted for compatibility with WEKA.

9.4 Feature Formation

This section presents the Feature Formation module used in this project. This module is responsible for creating the training and testing set. These sets are not usually split equally when performing supervised learning. As a rule one has to consider approximately $\frac{2}{3}$ of the sample set for training and the remaining $\frac{1}{3}$ for testing. In this case the data will be split by 67 %.

This module is quite similar to the one used in the last study, see [Larsen et al., 2014]. The difference is, that this module is not responsible for extracting the samples for the five chosen labels, since this is done in the previous modules.

The requirements to this module is that the loaded dataset and labels are of equal length. The dataset must consist of a multidimensional list or matrix, where each row denotes the features for each particular sample. The labels must be formatted as a list or vector, where each entry must fit to the row-entry of the dataset. If these requirements are not fulfilled, the training and testing data will not be split correctly and thereby fail the execution. If the execution does not fail, and the labels does not fit their corresponding row of features, the classification will be unreliable. The flow of the program can be seen in Figure 9.8.

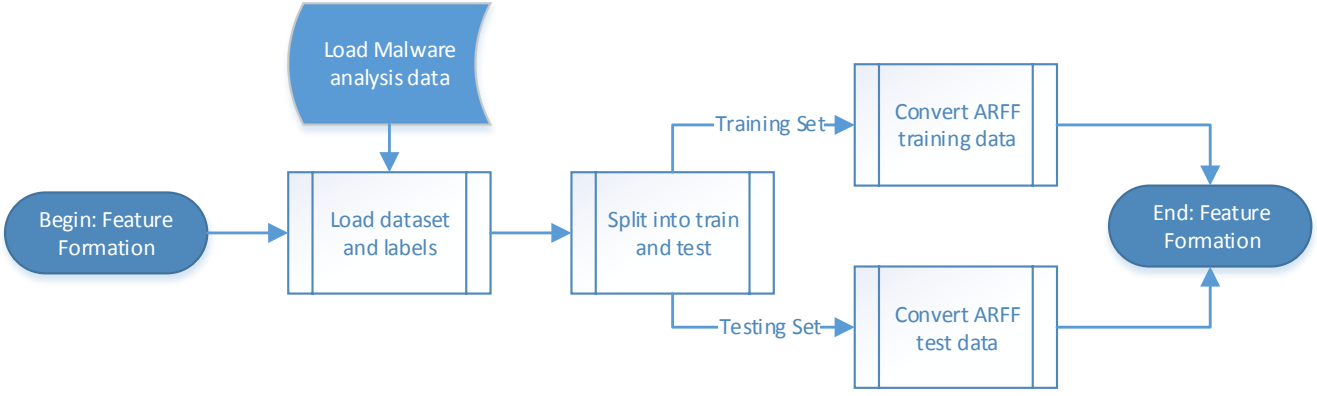


Figure 9.8: Flowchart of extraction of signature features.

As depicted in the flowchart, the sample set for the five classes are loaded from the generated MAT and P files. When this is done, the full sample set and the corresponding labels are split into training and

testing sets. The splitting function contains two major `for`-loops, where the first one sorts the samples according to the five classes. The structure consists of a nested list, where the m-dimension denotes the five classes and the n-dimension the corresponding feature data. These steps are performed on all five classes. The code, where these iterations are performed, can be seen in Code Snippet 9.9. Here, the construction of the sample set and label list are seen from Line 8 and down. The variables `slicedX` are the data and `slicedY` the labels.

Code snippet 9.9: Sorting the data according to the classes.

```
1  # For each unique label create sliced data
2  TrainingPercentage = .67
3  for ii in range(0, len(unique_Labels)): # Iterate through the set of classes
4    # For each label initialize next list
5    slicedX.append([])
6    slicedy.append([])
7    for jj in range(0, len(X)): # Iterate through dataset
8      if y[jj] == unique_Labels[ii]: # if label in set matches label in sampleset
9        slicedy[ii].append(y[jj]) # append label name
10       slicedX[ii].append(X[jj]) # append feature data
```

The second major `for`-loop splits the data equally between the classes, to their corresponding data set. This means that 67 % of the samples that contain the first class are split into the training set and the remaining 33 % of the same class split into the testing set. This is to secure, that the classes are balanced between the training and testing set, which will increase the "fairness" of the classification. An example of unfairly balanced data could be, if the training set contained 99 % for the first class and the rest of the classes had 50 % splits. For the first class, this leaves 1 % for the testing set, whereas other classes will have 50 %. When testing the classifier for class one, it has less evaluation data, which means the overall evaluation might not be valid. The code that splits the data equally between the classes is seen in Line 9.10.

Code snippet 9.10: Splitting the data equally between training and testing sets.

```
1  for ii in range(0, len(unique_Labels)): # Iterate through the set of classes
2    # Split data for each unique label according to percentage
3    X_train = X_train + slicedX[ii][0:int(len(slicedX[ii]) * TrainingPercentage)]
4    X_test = X_test + slicedX[ii][int(len(slicedX[ii]) * TrainingPercentage):int(len(slicedX[
         ii]))]
5    # Split labels for each unique label according to percentage
6    y_train = y_train + slicedy[ii][0:int(len(slicedy[ii]) * TrainingPercentage)]
7    y_test = y_test + slicedy[ii][int(len(slicedy[ii]) * TrainingPercentage):int(len(slicedy[
         ii]))]
```

The output from the process leaves four lists containing the data structure for the training set and the testing set, with the corresponding labels. The labels for both training and testing are then appended to the corresponding dataset, whereas two ARFF files are saved. The library `arff` is used to convert the created data structures to a file-format which is compatible in WEKA, see [Python Software Foundation, 2012]. Furthermore, the requirements in Chapter 8 are met.

The full code for this module can be found on the CD in the folder `Code\MultiClassClassification\`. This concludes the feature formation module, which is the last part of system design. The next part includes the results from the training and testing.

9.5 Chapter Summary

This chapter presented the overall system, along with descriptions of the Label Extraction, Feature Extraction and Feature Formation modules.

1. **System Overview**
   — The overall system was described, together with a brief summary of the Data Generation module.
2. **Label Extraction**
   — The label extraction for Microsoft labels was described.
3. **Feature Extraction**
   — The extraction and concatenation of features for sequence, frequency and signature representations were described.
4. **Feature Formation**
   — The concatenated features were split into 67 % training and 33 % testing sets, and converted to ARFF.

The next part will include the results for detection and classification.

INTENTIONALLY LEFT BLANK

Part IV

Results

INTENTIONALLY LEFT BLANK

CHAPTER 10

TRAINING

After designing and implementing the system, the predictive performance of the detection and family classification are evaluated. This section first introduces how Part IV is structured, i.e. the training and testing phase. The next two chapters will be organized as depicted in Figure 10.1.

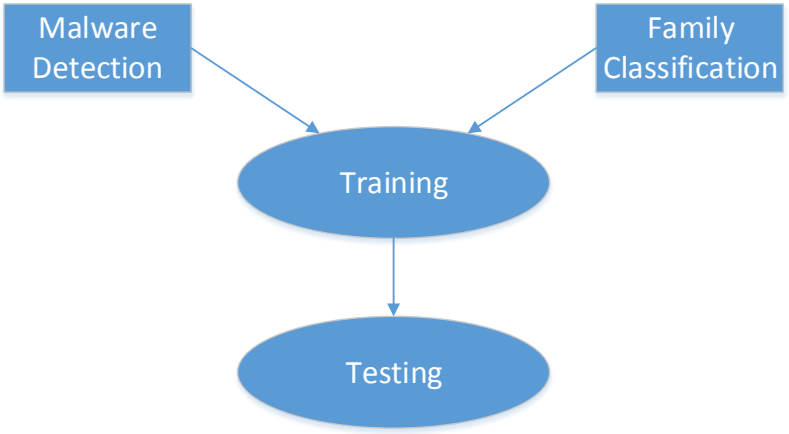


Figure 10.1: Structure of the training and testing phases.

Both malware detection and family classification include two feature representations, which are evaluated. Note, that the two representations in the family classification include 55 additional features. These are the family signatures. The complete list can be seen in Appendix A. Furthermore, the training set consists of 67 % of the data, while the testing set consists of the remaining 33 %.

Several preliminary tests have been made to clarify how many trees to use in the RF algorithm. In the last study, 160 trees were chosen, but this is not possible this semester due to hardware limitations. Tests were made between 1-160 trees and it was noticed when the algorithm reached around 50 trees, it switched to the SWAP-memory. The consequence of using the SWAP is the extreme decrease in R/W speed. This meant, that the experiment could run between 12 hours to several days, in which there was a potential risk of reaching the memory heap. If the heap was reached, the computer would crash, and the time it took to run the experiment would be wasted. From the successful experiments, only a 0.2 % increase in the amount of correctly classified instances, was seen from 50 to 100 trees. Because running with 50 trees in worst case, only took roughly 5 minutes and 100 trees took more than 24 hours, it was decided that 50 trees was the best choice.

This chapter evaluates the training models, i.e. the two feature representations, based on 10-cross validation. The measures used include: PPV, TPR and F-measure. A table will be shown, which include PPV and TPR, together with graphs, where the values are visualized. Afterwards, a second table will be presented with the F-measures. To increase readability of the results, each row is colored with different

scales of gray to rank the results. Here, gray is the best value and the whitest the worst. Each row is therefore independent from each other. The reason for depicting PPV and TPR, is because they directly express the relationships in the F-measure, see Section 7.5.

As mentioned earlier in Section 7.1, the dimensionality of the feature-space has to be reduced due to Hughes Effect. This of course assumes, that the features extracted from the database, are not flawless, i.e. some features might be redundant and irrelevant. Because of this fact, the goal of the training phase is to select the best features from the feature space. The feature selection method was chosen in Section 7.2. The chosen relevance measure is IGR and the algorithm used to rank the features, is Ranker. All the results and ARFF files from the training and testing phase, can be found in their respective folders on the CD `Results\`.

10.1 Malware Detection

This section consist of two parts, one for each representation, done in order to keep the results separated. The different features and their amount for both representations were described in Subsection 6.4.1.

The next two parts present the results for Representation 1 and 2, respectively. Each part includes the results for FS, where the sample set includes all 837 cleanware and 1,000 malware. Because the samples are split into training and testing sets, the training will here consist of 560 cleanware and 670 malware. Each malware is from a different family. After presenting the results, a number of features for FS are chosen, based on the best performance. The chosen FS is afterwards used in an additional test with the goal of deciding the number of malware samples to use in training. This is done in order to test the effects of a non-uniform distribution of samples between cleanware and malware, see Section 3.1.

10.1.1 Representation 1

Below in Table 10.1, the results for FS are listed. The test is made using 10-fold cross validation using 10, 50, 150, 250, 350 and full feature set for comparison.

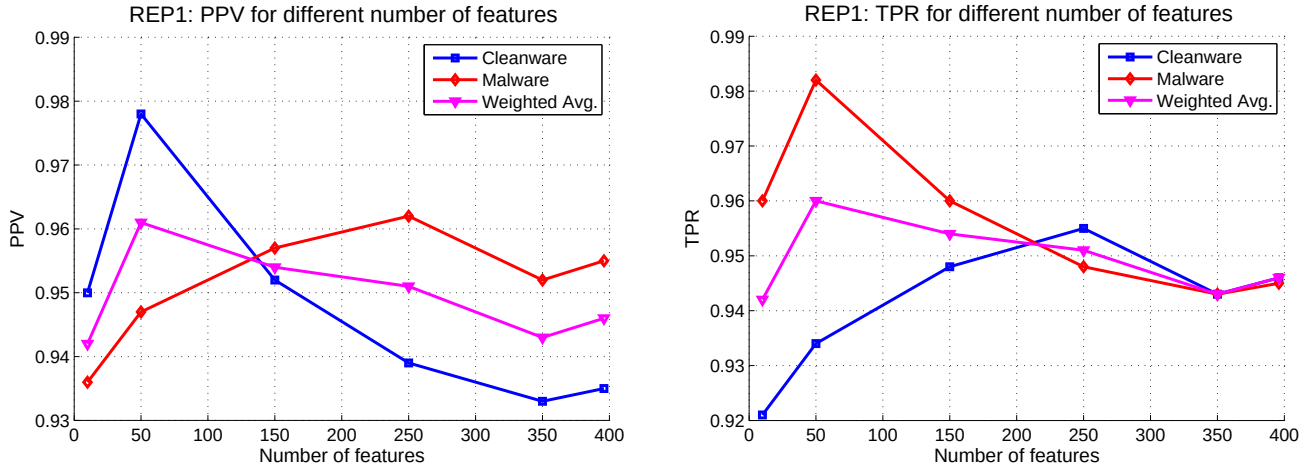
Rep1 Classes	**Metrics**	**Number of features**					
		10	**50**	150	250	350	396 (all)
Cleanware	PPV	0.950	0.978	0.952	0.939	0.933	0.935
	TPR	0.921	0.934	0.948	0.955	0.943	0.946
Malware	PPV	0.936	0.947	0.957	0.962	0.952	0.955
	TPR	0.960	0.982	0.960	0.948	0.943	0.945
Weighted avg.	PPV	0.942	0.961	0.954	0.951	0.943	0.946
	TPR	0.942	0.960	0.954	0.951	0.943	0.946

Table 10.1: PPV and TPR for different amount of features according to the FS, where the gray-scale denotes how good the value is, relative to the rest of the row.

From Table 10.1 it is noticed, that the highest PPV and TPR are seen at 50, 150 or 250 features. The objective for this training, is to find the best performing number of features, but at the same time also

secure a high detection rate for Malware. It is seen that cleanware has the highest PPV at 50 features, whereas the highest TPR is seen at 250 features.For Malware, the highest PPV is seen at 250 features, and the highest TPR at 50 features. This intuitively makes sense, since the classifier is binary. This means that a high FN for cleanware, gives a high FP for malware and vice versa.

The values in Table 10.1 (p. 82) are also visualized in Figure 10.2a and Figure 10.2b. Using the graphs, a better overview of how the FS influences the performance, is seen. FS with 50 and 150 features are the best candidates if the classifier's performance for malware is prioritized. For this project it has been decided, that it is most favorable to detect as much malware (low FN) as possible, at the expense of FPs. This can be reasoned, since a FN is a miss, thus the classifier is not detecting the malware, which can be more costly than the opposite case. Therefore, the focus is to achieve as few FNs as possible, meaning a high TPR for malware, but not with a too large negative impact on PPV.



(a) PPV.

(b) TPR.

Figure 10.2: PPV and TPR for each of the classes along with the weighted average.

The overall performance of the classifier for PPV seems unstable. Looking at the weighted average, it is observed that the PPV at 10 and 350 features overall are the lowest, whereas the full feature set performs better. Studying the results at 50, 150 and 250 features it is concluded, that FS has a positive influence of the performance.

Looking at the graphs for TPR, again instability are observed. 10 features still provides the worst results for both classes, whereas 50, 150 and 250 features, again performs better. The weighted average results show, that FS with 50 and 150 features provide the best detection.

From the figures, 150 features seem to give a more even performance of the classifier for both cleanware and malware. Here, 50 features provide a good performance of the classifier for malware's TPR, but because cleanware has many FNs, indicated by the low TPR, the performance of malware's PPV is lower. Since the performance of malware for TPR is prioritized, 50 features are the best choice. Of course, choosing 150 features increases the performance of malware's PPV and cleanware's TPR. At the same time it introduces more FNs for malware, which means that the classifier's ability to detect, is impaired.

This can be explained if the 50 features chosen by FS, are better to discriminate malware from cleanware, whereas introducing more features, makes the classifier confuse the classes more frequently. Choosing 50 features, the classifier achieves ≈ 0.2 higher TPR, which is quite significant, but only ≈ 0.1 less PPV. This is why 50 features are chosen.

As a conclusion, the F-measure is used, which can be found in Table 10.2. The F-measure summarize both the PPV and TPR by one measure, in which it favors the measure that shows the worst performance of the two.

Rep1 Classes	**Metric**	**Number of features**					
		10	**50**	150	250	350	396(all)
Cleanware	F-M	0.936	0.955	0.950	0.947	0.938	0.941
Malware	F-M	0.948	0.964	0.958	0.955	0.948	0.950
Weighted avg.	F-M	0.942	0.960	0.954	0.951	0.943	0.946

Table 10.2: F-measure for different amount of features according to FS, where the highlighting denotes how good the performance is on a row basis.

From the table it is clear that FS with 50 features show the best performance, followed by 150 and 250 features. The worst performance is seen when 10 and 350 features are selected. Studying these results, it is concluded that there is a gain in performance when FS is used.
A list of the 50 features chosen by FS, where the top being the best, can be found in Appendix D (p. 131). Furthermore, a list translating ID number of the API into its name can be found in Appendix C. Looking at the features chosen, it is noticed that most of the features are the API frequencies, followed by the frequency for DLL files, two frequency bins and at last one sequence number with argument. This indicates, that the frequencies of the different API calls are good to discriminate between cleanware and malware when trained using only 670 malware samples. The top 3 features include, the frequency of API `ExitProcess`, API `NtOpenDirectoryObject` followed by the first sequence with argument for each sample. Studying API `ExitProcess`, it is noticed, that it is used for existing a process in Windows, as the name indicates. It is believed, that this API can be used to discriminate the classes, if malware tries to exit its processes after infecting the system, in order to hide itself. Looking at the second best feature, it is a function used to open object directories (not to be confused with file system directories), [Microsoft, 2015e] [Microsoft, 2015f]. From [Blasco, 2009], it is seen that mutexes are located in directories, indicating that the malware might use this API in combination with others, to search for mutexes. The third feature, indicates that the first API call with its argument, can give a trace to which class the samples belong to.

It was decided to use FS with 50 features. Additionally a new training phase for Representation 1 is carried out to determine how many malware samples to include in the testing phase.

Selection of the amount of malware samples

The amount of malware samples to include is decided using FS with 50 features. This is needed, since the impact of a sample set being non-uniform has to be tested. Because the amount of cleanware samples are

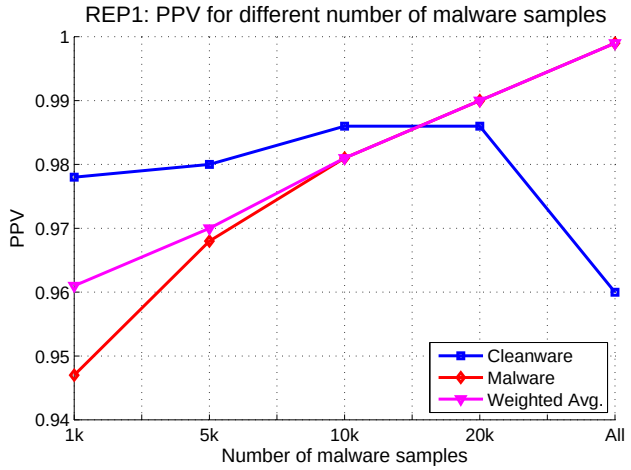
low, it is needed to only add a fraction of the malware samples to the sample set. The classifier should include as many different families and as many samples possible. Since the sample set for malware, in this project contains over 6,000 different families, this goal is difficult to fulfill. A sample set is created containing malware samples with 67 % of 1,000, 5,000, 10,000, 20,000 and ≈ 175,000 (all), see Subsection 9.3.4.

The results from the training are listed in Table 10.3.

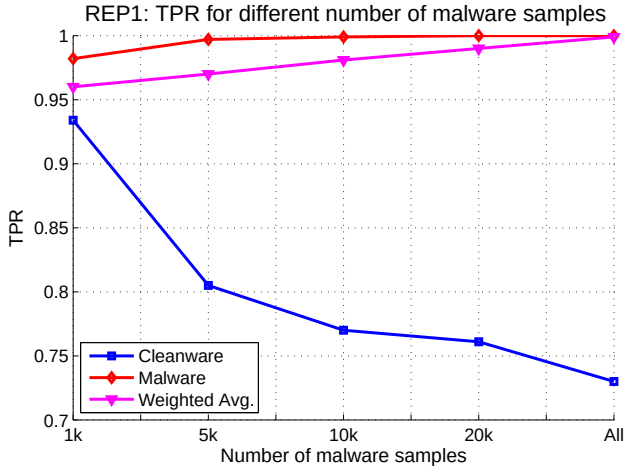
Rep1 Classes	**Metrics**	**Number of malware samples**				
		1k	**5k**	10k	20k	All
Cleanware	PPV	0.978	0.980	0.986	0.986	0.960
	TPR	0.934	0.805	0.770	0.761	0.730
Malware	PPV	0.947	0.968	0.981	0.990	0.999
	TPR	0.982	0.997	0.999	1.000	1.000
Weighted avg.	PPV	0.961	0.970	0.981	0.990	0.999
	TPR	0.960	0.970	0.981	0.990	0.999

Table 10.3: PPV and TPR for different amount of malware samples according to the FS of 50, where the gray highlighted values denotes the max. value.

It is noticed that detection is nearly perfect when all malware samples are included, but with significant loss in PPV and TPR for cleanware. Table 10.3 shows a tendency, where it is observed, that as more malware are included, the classifier becomes better at detection. At the same time, this gives rise to more FPs for malware, which is why the classification of cleanware degrades. Even though FPs for malware are undesired, it is a good sign, that the performance when detecting cleanware, is not worse. This is argued as only 151 cleanware are falsely classified as malware, out of a total of 560, when comparing to ≈ 117,000 malware samples.
The table is visualized in Figure 10.3a and in Figure 10.3b. It is clear that the performance for malware increases with the amount of samples that are included.



(a) PPV.



(b) TPR.

Figure 10.3: PPV and TPR for each of the classes along with the weighted average.

Note, that the PPV slightly increases for cleanware as the malware samples are increased, after which it decreases again after 20k. Looking at TPR for cleanware, it can clearly be seen, that the performance decreases, since the number of FNs increases for cleanware, while the samples increase. The performance when looking at both classes is best at 1k and 5k samples, whereas the PPV and TPR both increase for malware from 1k to 5k. The same picture is seen for cleanware's PPV.

Since the objective is to detect as much malware as possible, the decision is made to use 5k samples. It is believed, that the PPV and TPR increased for malware, since a lot more samples were include to in sample set. The PPV for cleanware also increased, while the amount of samples for cleanware remained the same, which is positive. Of course, 10k samples were also a potential candidate, since the number of FPs for cleanware decreases even more. Due to the decreased uniformity along with a TPR reaching ≈ 0.75 for cleanware, 10k samples are a less attractive choice.

A summary of the results are found in Table 10.4, showing the F-measure for training.

Rep1 Classes	**Metric**	**Number of malware samples**				
		1k	**5k**	10k	20k	All
Cleanware	F-M	0.955	0.884	0.865	0.859	0.830
Malware	F-M	0.964	0.983	0.990	0.995	0.999
Weighted avg.	F-M	0.960	0.969	0.980	0.989	0.998

Table 10.4: PPV and TPR for different amount of malware samples according to the FS of 50, where the highlighting denotes how good the performance was on a row basis.

Note, that the F-measure decreases steadily for cleanware, while steadily increases for malware. Judging by the table, it seems that 10k is the better choice, if the results should be balanced between the two. Though, 5k samples give better TPR for cleanware compared to 10k and is therefore chosen. It can be argued that 1k is better, since it is uniform, but because the objective is to detect as much malware as possible and still maintain a small amount of FPs for Malware, 5k is chosen.

10.1.2 Representation 2

This part follows the exact same structure as the first part, but also compares the results for the representations. Therefore, the description of the experiment is not repeated again. However, it should be clear that more features are present for Representation 2, and therefore include more tests, as the same interval between the FS is used.

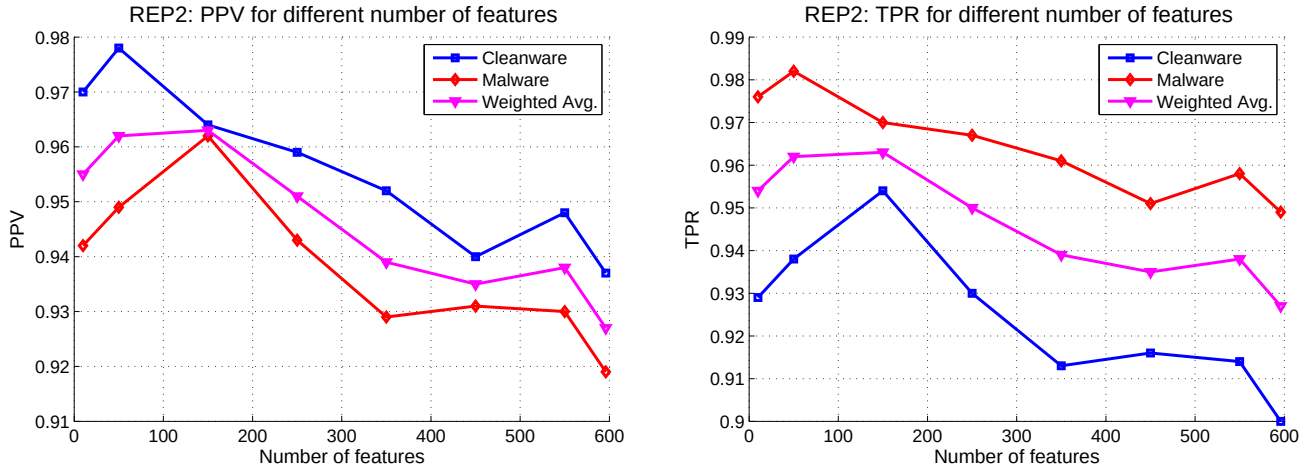
The results of the training with different amount of features used in the FS, can be seen in Table 10.5 (p. 87). It is noticed, that the results are best between 10 to 250 features, as was also seen for Representation 1. A slight difference is, that 10 features, provide better results compared to Representation 1. This could be caused by the fact that separating sequence from its arguments, might give better discrimination.

Rep2 Classes	Metrics	Number of features							
		10	**50**	150	250	350	450	550	596(all)
Cleanware	PPV	0.970	0.978	0.964	0.959	0.952	0.940	0.948	0.937
	TPR	0.929	0.938	0.954	0.930	0.913	0.916	0.914	0.900
Malware	PPV	0.942	0.949	0.962	0.943	0.929	0.931	0.930	0.919
	TPR	0.976	0.982	0.970	0.967	0.961	0.951	0.958	0.949
Weighted avg.	PPV	0.955	0.962	0.963	0.951	0.939	0.935	0.938	0.927
	TPR	0.954	0.962	0.963	0.950	0.939	0.935	0.938	0.927

Table 10.5: PPV and TPR for different amount of features according to the FS, where the gray highlighted values denotes the max. value.

For 10, 50 and 150 features, a better TPR for cleanware is noticed compared to what was seen for Representation 1. At the same time it also maintains a good PPV. The tendency is the same for malware, where PPV and TPR have also increased. Most gain in performance is seen at 10 features. The first 10 features used, only contain one sequence feature, namely sequence 1. This feature was also used in Representation 1, as the second best feature, but here it is the fourth best. This feature might give a better discrimination compared to the combination of sequence with argument. Furthermore, it is noticed that only a couple of frequency APIs are switched with new APIs. These changes, might be the reason, that the discrimination is better, and thus provide better results. Again, 50 and 150 features provide the best results as was also the case for Representation 1.

The table is visualized in two figures, namely Figure 10.4a and Figure 10.4b, which overall shows a more steady decrease in performance as the amount of features increases.



(a) PPV.

(b) TPR.

Figure 10.4: PPV and TPR for each of the classes along with the weighted average.

The same question from Representation 1 can be asked: Should FS with 50 or 150 features be chosen? It is noticed, that the PPV for malware increases from 50 to 150 features, whereas the PPV decreases for cleanware. Looking at the TPR, it decreases for malware, while it increases for cleanware. This project prioritize the detection of as malware as possible, i.e. a high TPR is favored for malware, assuming that

the PPV is high as well. Since PPV for malware only decreases ≈ 0.1 from 150 to 50 features, 50 features are chosen again. In return, TPR increases from 0.97 to 0.982, thus less amount of FNs for malware are present. Viewing the results for cleanware at 50 features, the results are satisfactory, with 0.978 in PPV and 0.938 in TPR.

Below in Table 10.6, the F-measure is seen as a conclusion of the results.

Rep2 Classes	**Metric**	**Number of features**							
		10	**50**	150	250	350	450	550	596(all)
Cleanware	F-M	0.949	0.957	0.959	0.945	0.932	0.928	0.931	0.918
Malware	F-M	0.959	0.966	0.966	0.955	0.945	0.941	0.944	0.934
Weighted avg.	F-M	0.954	0.962	0.963	0.950	0.939	0.935	0.938	0.927

Table 10.6: F-measure for different amount of features according to the FS, where the highlighting denotes how good the performance was on a row basis.

This time the F-measure actually points at FS with 150 features as the best candidate, but simply because of the higher amount of FNs for malware, at 150 features, the decision of 50 remains. The difference in F-measure between the 50 and 150 features is also minimal.

Looking at the 50 features chosen by FS for Representation 2, see Appendix D, it is again noticed, that the frequency of the APIs are mostly used. In comparison, more frequency bins are chosen with a total of 5. Also, 5 frequency DLLs and 1 sequence, here the first one in the sequence representation, have been chosen. The top 3 are different in comparison to Representation 1, namely `ExitProcess`, `NtFreeVirtualMemory` and `NtOpenDirectoryObject`. Two of the APIs from Representation 1 have already been discussed, whereas API `NtFreeVirtualMemory` was not seen. It was found that this API has the same function as the name indicates, and is mostly used by malware, hence making it more discriminative.

Since it was decided to use FS with 50 features, a new training phase for Representation 2, is carried out to determine the number of malware samples to include in the testing phase.

Selection of the amount of malware samples

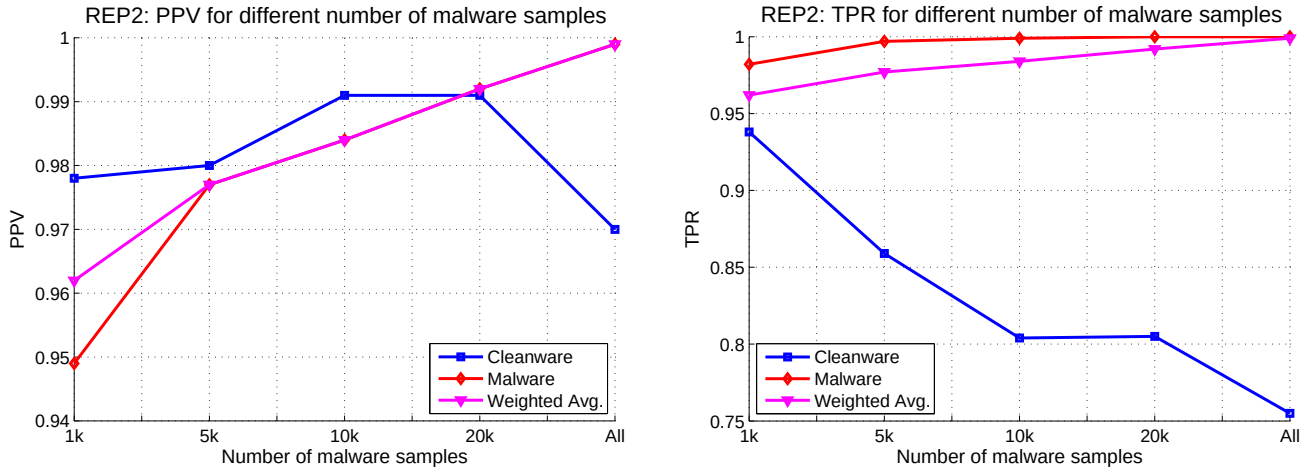
The results for this training phase can be found in Table 10.7 (p. 89). Again, as for Representation 1, the same trend of achieving better results for malware as the amount of samples increase, is seen. Though, better results for cleanware when looking at the TPR are seen.

Rep2 Classes	Metrics	Number of malware samples				
		1k	**5k**	10k	20k	All
Cleanware	PPV	0.978	0.980	0.991	0.991	0.970
	TPR	0.938	0.859	0.804	0.805	0.755
Malware	PPV	0.949	0.977	0.984	0.992	0.999
	TPR	0.982	0.997	0.999	1.000	1.000
Weighted avg.	PPV	0.962	0.977	0.984	0.992	0.999
	TPR	0.962	0.977	0.984	0.992	0.999

Table 10.7: PPV and TPR for different amount of malware samples according to the FS of 50, where the gray highlighted values denotes the max. value.

This time, the decrease in results for cleanware by including 5k instead of 1k samples, is not that significant as for Representation 1. Though, malware still performs better, since it has more samples. In general the results have increased in performance compared to Representation 1.

The results are visualized in Figure 10.5a and Figure 10.5b, showing the same tendency as for Representation 1.



(a) PPV.

(b) TPR.

Figure 10.5: PPV and TPR for each of the classes along with the weighted average.

It is noticed that the TPR for cleanware is not decreasing significantly from 1k to 5k, which was the case for Representation 1. By this observation, it is believed that Representation 2 will perform better in the testing phase. Looking at malware, the PPV has increases by more than 0.1, whereas the TPR remains the same. As before, it has been chosen to use 5k malware samples.

To summarize the result, the F-measure is depicted in Table 10.8 (p. 90).

Rep2 Classes	Metric	Number of malware samples				
		1k	**5k**	10k	20k	All
Cleanware	F-M	0.957	0.915	0.888	0.889	0.849
Malware	F-M	0.966	0.987	0.992	0.996	0.999
Weighted avg.	F-M	0.962	0.977	0.984	0.992	0.999

Table 10.8: PPV and TPR for different amount of malware samples according to the FS of 50, where the highlighting denotes how good the performance was on a row basis.

Almost the same tendency from Representation 1 is seen again. A step-wise increase and decrease in results for malware and cleanware are seen respectively. As detection for malware is prioritized, 5k is still chosen, even though a large decrease in F-measure is seen for cleanware, which is justified by the increase of 0.2 in F-measure for malware.
It is known that increasing the amount of malware samples, might produce better results for malware, since more FPs and FNs need to be present to decrease the PPV and TPR. This assumes, that the amount of FPs and FNs does not increase linearly with the amount of malware samples. In fact, it is verified that the amount of FNs decrease for malware when more samples are added, while its FPs increased. Last mentioned causes the TPR for cleanware to decrease. At last, the increase in malware samples, shows that the classifier can handle 5 times more malware and still produce less amount of FNs, where it only misses 10 malware samples out of 3,350. This also helps to illustrate, that the method and features chosen, can in fact discriminate malware and cleanware.

10.2 Family Classification

This section is divided in two, where the first part discusses the results from Representation 1 and second part Representation 2. The different features and their amount for both representations were described in Subsection 6.4.2.

The total amount of samples in the dataset is 31,295. The distribution of the malware samples for training and testing is depicted in Figure 10.6.

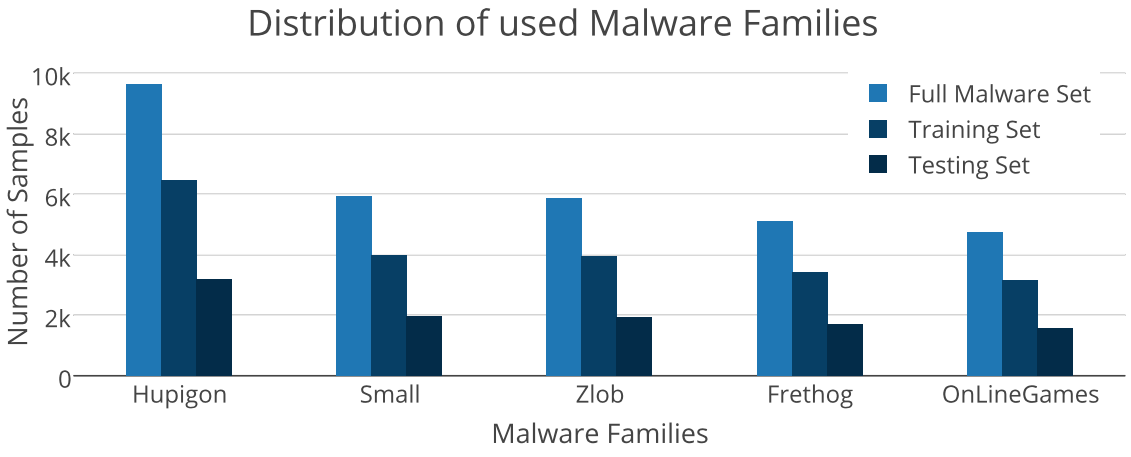


Figure 10.6: Distribution of the malware families in the family classification.

10.2.1 Representation 1

The chosen amount of features in FS are the following: 10, 50, 100, 200, 300, 400 and 451(all). All features are included to verify the assumption that FS can be used to filter redundant or irrelevant features. This means, that it is expected to see a drop in the predictive performance using all the features compared to the best performing feature reduction. This is seen when studying the results in Table 10.9.

Rep1 Classes	**Metrics**	**Number of features**						
		10	50	100	200	300	**400**	451(all)
Small	PPV	0.938	0.934	0.919	0.896	0.891	0.891	0.887
	TPR	0.553	0.701	0.780	0.782	0.776	0.782	0.777
OnLineGames	PPV	0.770	0.910	0.929	0.927	0.929	0.930	0.929
	TPR	0.021	0.489	0.688	0.722	0.726	0.726	0.727
Hupigon	PPV	0.916	0.761	0.853	0.904	0.902	0.906	0.902
	TPR	0.597	0.947	0.973	0.966	0.963	0.963	0.962
Frethog	PPV	0.987	0.929	0.657	0.803	0.805	0.724	0.725
	TPR	0.293	0.589	0.918	0.762	0.763	0.904	0.905
Zlob	PPV	0.293	0.520	0.894	0.731	0.731	0.852	0.851
	TPR	0.992	0.816	0.704	0.906	0.907	0.834	0.830
Weighted avg.	PPV	0.793	0.800	0.853	0.857	0.856	0.867	0.865
	TPR	0.526	0.748	0.834	0.850	0.848	0.859	0.857

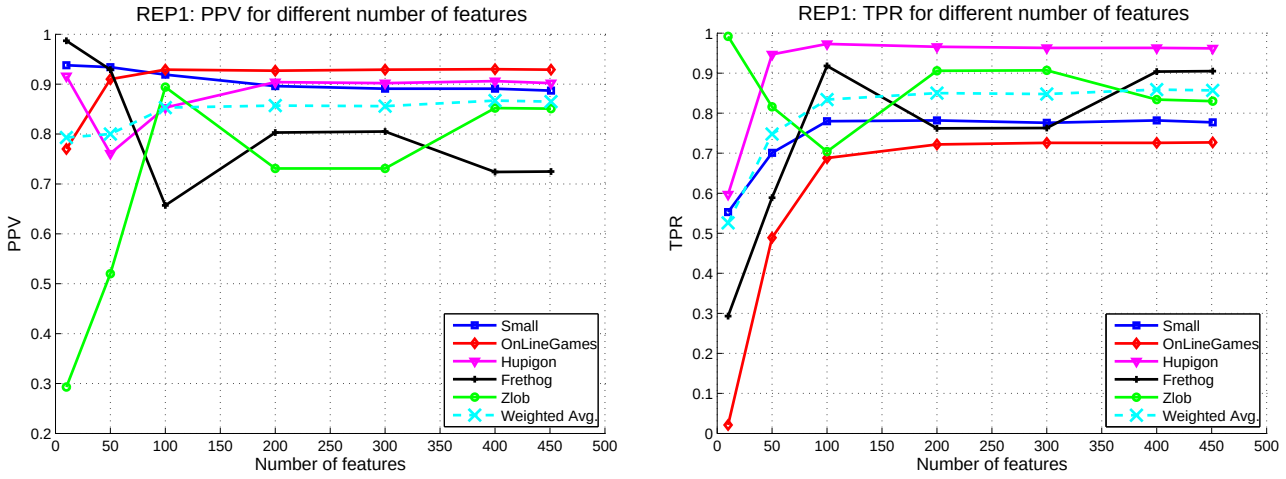
Table 10.9: PPV and TPR for different amount of features according to the FS, where the gray-scale denotes how good the value is, relative to the rest of the row.

Here the PPV and TPR results are shown for each of the classes accordingly. The largest concentration of high values lie between 200 and 451 features. Studying the difference between the values at each row, it is noticeable that the best overall prediction, is when using 400 features. This is also what the weighted average indicates, which is based on the sample size for each of the classes.

For better overview, the PPV and TPR are plotted in Figure 10.7a (p. 92) and Figure 10.7b (p. 92). Observe here, the aggressive fluctuations between 10 and 200 features. This might indicate, that the classifier is not yet stable in its prediction, thus the impact of adding new features will yield very beneficial for one class, while very damaging to another. This behavior is most clear for **Frethog** and **Zlob**, where their graphs seem to have a reverse relationship. More in depth analysis between the behavior of the classes are described in the testing phase.

Studying the two figures, the performance stabilizes at around 400 features, while **Frethog** and **Zlob** still tend to fluctuate. Analyzing Figure 10.7a (p. 92) it is seen that, even though the performance of **Frethog** drops at 400, the increase for **Zlob** is greater. This is also why the weighted average is larger at 400 compared to 300. Note here, that the PPV describes the relationship between the TPs and FPs, which means that the high values express a low amount of FPs. At last, it is also verified that the performance drops from 400 to 451 features, even though the drop is not significant. This means that

a number of irrelevant or redundant features are present in the feature space. Including features that will add zero discrimination between the classes will degrade the performance of the classifier due to Hughes Effect. The reasoning for this can also be traced back to how the RF algorithm works. Because RF randomly picks out m features from the whole feature space according to $\mathbf{int}\log_2(M)+1$), as you increase the amount of irrelevant features, there will be a higher probability of picking poor features to split the nodes.



(a) PPV.

(b) TPR.

Figure 10.7: PPV and TPR for each of the classes along with the weighted average.

Analyzing the TPR in Figure 10.7b, the same fluctuations are present, but some of them are close to reversed compared to the PPV. This is because TPR concerns the relationship between the TPs and FNs, whereas PPV concerns the relationship between TPs and FPs. The amount of FPs and FNs are imbalanced for each individual class, which is why the graphs for both PPV and TPR are not the same. Note, that the amount of TPs are the same in both calculations. Again the graphs tend to stabilize at around 400 features, while **Frethog** and **Zlob** still fluctuate slightly. The increase for **Frethog** from 300 to 400 is larger than the decrease of **Zlob** in the same interval, which means that choosing a 400-dimensional feature space is best.
To conclude, Table 10.10 shows the F-measure.

Rep1 Classes	**Metric**	**Number of features**						
		10	50	100	200	300	**400**	451(all)
Small	F-M	0.695	0.801	0.844	0.835	0.830	0.833	0.828
OnLineGames	F-M	0.041	0.636	0.791	0.812	0.815	0.815	0.816
Hupigon	F-M	0.723	0.844	0.909	0.934	0.931	0.934	0.931
Frethog	F-M	0.452	0.721	0.766	0.782	0.783	0.804	0.805
Zlob	F-M	0.453	0.642	0.788	0.809	0.810	0.843	0.840
Weighted avg.	F-M	0.520	0.764	0.833	0.849	0.847	0.858	0.857

Table 10.10: F-measure for different amount of features according to the FS, where the highlighting denotes how good the performance was on a row basis.

This measure also shows a high concentration of high values from 200-451 features, but compared to Table 10.9 (p. 91), the results are less ambiguous, and therefore it is more clear that 400 is the best number of features. It is also noticed again, that there is a drop in the performance from 400 to 451, even though it is not significant. To support the results and the choice, the feature ranking list has also been studied. According to the IGR values, which are calculated for each of the features, only 388 of them will contribute to the classification. This might explain the slight drop in performance when validating the full feature set, because of the RF algorithm.

This concludes the Family Classification training phase for Representation 1. This representation is used with the **400** most discriminative features for final testing in Section 11.2.

10.2.2 Representation 2

This subsection discuss the results for Representation 2. The instances are the following: 10, 50, 100, 200, 300, 400, 500, 600 and 651(all). In Representation 1, the performance dropped when using all the features compared to 400 even though the drop was not significant. A drop in the performance is also expected for Representation 2, when using all features compared to the best performing feature reduction. This is noticed when studying the results in Table 10.11.

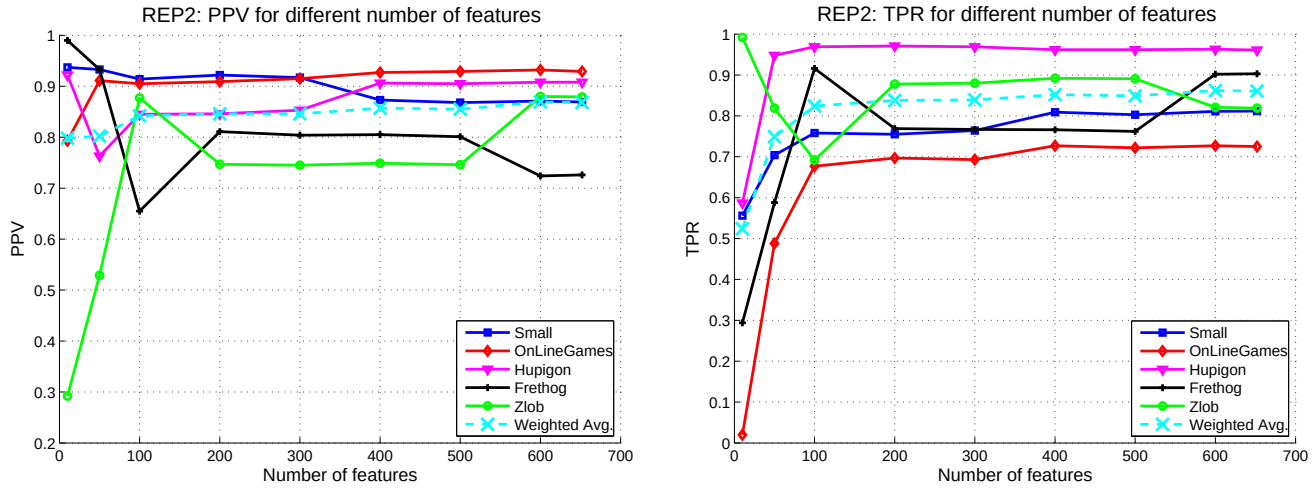
Rep2 Classes	**Metrics**	**Number of features**								
		10	50	100	200	300	400	500	**600**	651(all)
Small	PPV	0.937	0.933	0.914	0.922	0.917	0.873	0.868	0.871	0.869
	TPR	0.556	0.704	0.758	0.755	0.764	0.809	0.803	0.811	0.812
OnLineGames	PPV	0.793	0.911	0.905	0.909	0.915	0.927	0.929	0.932	0.929
	TPR	0.020	0.488	0.677	0.697	0.693	0.727	0.722	0.727	0.725
Hupigon	PPV	0.922	0.763	0.845	0.846	0.853	0.906	0.905	0.908	0.908
	TPR	0.587	0.948	0.969	0.971	0.969	0.962	0.962	0.963	0.961
Frethog	PPV	0.990	0.933	0.655	0.811	0.804	0.805	0.801	0.724	0.726
	TPR	0.294	0.588	0.916	0.769	0.767	0.766	0.762	0.902	0.903
Zlob	PPV	0.292	0.529	0.877	0.747	0.745	0.749	0.746	0.880	0.879
	TPR	0.992	0.819	0.692	0.878	0.880	0.892	0.891	0.821	0.819
Weighted avg.	PPV	0.798	0.802	0.842	0.846	0.846	0.857	0.855	0.869	0.868
	TPR	0.524	0.749	0.824	0.838	0.839	0.852	0.849	0.862	0.861

Table 10.11: PPV and TPR for different amount of features according to the FS, where the highlighting denotes how good the performance was on a row basis.

Viewing the results, it is seen that the highest concentration of high values lie between 200 and 651(all) features. Studying the difference between the values at each row, the best overall prediction is when using 600 features. This is also indicated by the weighted average. Just like before, to clarify this observation, the PPV and TPR are plotted in Figure 10.8a (p. 94) and Figure 10.8b (p. 94). It is observed, that the behavior is similar to what was shown in Subsection 10.2.1, thus displaying quite aggressive fluctuations in the beginning. Again, as more features are included, the balance between the FPs and FNs stabilizes.

As before, **Frethog** and **Zlob** seem to have a reverse relationship, which could mean that these two families are most sensitive to changes in the features.

The graphs tend to stabilize at around 600 features. Note here, that it cannot be sure if the graphs have actually converged. If more features had been extracted, e.g. 1000 more features, the graphs might have converged to the class probabilities. This observation was described by Gordon F. Hughes in [Hughes, 1968]. Analyzing Figure 10.8a it is seen, that even though **Frethog** drops at 600, the increase in the performance for **Zlob** is greater. This was also the case in Representation 1, but at 400 features. This means that, also in Representation 2 there are a number of redundant or irrelevant features, present in the feature set. With the same reasoning as before, including features that contain no discriminative power will degrade the performance of the classifier, due to Hughes Effect.



(a) PPV.

(b) TPR.

Figure 10.8: PPV and TPR for each of the classes along with the weighted average.

Viewing the TPR in Figure 10.8b, the same fluctuations are present, and as for Representation 1, some of them are close to reversed. This is because, those classes that had very few FPs, in contrast has a significant amount of FNs. There is somewhat an equal balance between the FPs and FNs for **OnlineGames** and **Small**. Furthermore, these classes does not fluctuate as wildly as the rest. In this figure, it is also noticed, that while **Zlob** decreases from 500 to 600 features, the increase in **Frethog**, at the same instances, is larger. This means that the optimal choice of feature space is 600.

To conclude, Table 10.12 (p. 95) shows the F-measure at the same feature instances as before. It is seen, that the highest concentration of dark gray is at 600 features. This is also verified by the weighted average, where the sample size of each class is taken into account.

Rep2 Classes	Metrics	Number of features								
		10	50	100	200	300	400	500	**600**	651(all)
Small	F-M	0.698	0.803	0.829	0.831	0.834	0.840	0.834	0.840	0.840
OnLineGames	F-M	0.040	0.635	0.775	0.789	0.789	0.815	0.812	0.816	0.815
Hupigon	F-M	0.718	0.846	0.903	0.904	0.907	0.933	0.933	0.935	0.934
Frethog	F-M	0.453	0.721	0.763	0.790	0.785	0.785	0.781	0.803	0.805
Zlob	F-M	0.451	0.643	0.774	0.807	0.807	0.814	0.812	0.849	0.848
Weighted avg.	F-M	0.518	0.747	0.822	0.836	0.837	0.851	0.848	0.861	0.861

Table 10.12: F-measure for different amount of features according to the FS, where the highlighting denotes how good the performance was on a row basis.

The results show a less ambiguous picture compared to Table 10.11 (p. 93), which also supports the decision of using the 600 best features according to IGR. Studying the IGR for each of the features, it is noticed that according to the algorithm, 588 out of the 651 features yield a information gain. This means, that the remaining features actually contributed with zero discrimination. This fits nicely with the results, where a noticeable, but not significant, drop is observed from 600 to 651. Because the weighted average is actually the same, the full feature set could also be used, but assuming that Hughes Effect is present, it has been decided to use 600 features in the testing phase.

10.3 Chapter Summary

This chapter presented the results from the training phase. The goal of the training phase was to first find the best suitable amount of malware samples that should be used for testing malware detection. Hereafter, the goal was to find the best performing set of features using FS, for both representations in both detection and family classification. A summary of the main findings are the following:

1. **Malware Detection**
   — FS with 50 features and a 5k sample set yielded the best performance for both representations.
2. **Family Classification**
   — FS with 400 features yielded the best performance for Representation 1, whereas 600 features were best for Representation 2.

The next chapter presents the testing phase for both malware detection and family classification. Here both representation methods are tested to validate if the performance are the same as the training phase.

INTENTIONALLY LEFT BLANK

CHAPTER 11

TESTING

For testing, the chosen FS is applied on the testing set, where a mapping function is used to map the features together. This secures, that the same features used in the training phase, are also used in the testing phase. Two subsections are presented, where the results from Representation 1 are first discussed followed by the discussion for Representation 2.

In both subsections, a table including the results with relevant performance metrics are shown. The metrics include TPR, FPR, PPV F-M and AUC. The results in the table will be colored in gray scale on a **column** basis. This also means, that the color of the columns are independent from each other and should only be read **column by column**. The performance for each of the classes are discussed, supported by the weighted average. To give a better overview of how the classifier predicts each instance, a confusion matrix are also presented. At last, the overall performance is evaluated by showing the ROC curves for each of the classes.

There will be no summary in this chapter, but instead a discussion and conclusion after both detection and family classification.

11.1 Malware Detection

This section includes the test results for malware detection. The models created in the training phase for both Representation 1 and 2 include 50 features. The test set includes 277 cleanware and 1,650 malware samples, which is 33 % of the total sample set used for malware detection. After presentation and discussion of the results, an analysis is done on the used features, during the testing.

11.1.1 Representation 1

The results can be found in Table 11.1. It should be emphasized, that the two classes are not uniform, which might have an impact on the results, and thus needs to be taken into account. After presenting both representations a discussion and assessment of the classifiers usability are made.

Class	TPR	FPR	PPV	F-Measure	AUC
Cleanware	0.805	0.003	0.978	0.883	0.994
Malware	0.997	0.195	0.968	0.982	0.994
Weighted Avg.	0.969	0.167	0.970	0.968	0.994

Table 11.1: Test results for Representation 1.

Cleanware, as expected from the training phase, has a quite low TPR of 0.805. This is caused by malware having a relatively high amount of FPs, since it is a binary classifier. This means, that the

performance of one class affects the other class directly. The FPR for cleanware is very low, thus the amount of FPs are low compared to its amount of TNs. Therefore, a high PPV of 0.978 for cleanware is seen, also indicating that the detection of malware must be good. Since the TPR is low for cleanware, the F-measure only scores 0.883. Comparing the F-measure with the training results, it is noticed that it is close to having the same performance, only 0.001 lower. The AUC value is high but since the data is imbalanced, AUC can be imprecise.

Malware performs well, with a high TPR of 0.997, but introduces a high FPR of 0.195. Since the cleanware sample set contains few samples, the number of TNs for malware is quite low. This means, that FPs for malware have a higher impact on the FPR, see Equation 7.10. The FPs also affect the PPV, but in this case not by much, though it is still lower than cleanware. The high PPV for cleanware is reasonable, since the high TPR for malware means that the amount of FPs for cleanware are low. Even though, the PPV for malware is lower than for cleanware, the F-measure for malware is quite good, here 0.982. Again, the AUC is high, and is actually the same as cleanware, but because there is an imbalance in the dataset, the effects on cleanware's FPR, are quite small. The reason is, that there is a high number of TNs (TPs for Malware) in the denominator compared to FNs.

Looking at the weighted average, the overall performance for the classifier is satisfactory compared to the goal of this study. It performs good in both TPR and PPV, but with a quite high FPR. This also results in a reasonable F-measure of 0.968, but this measure hides the fact that the TPR for cleanware is low. Since a high TPR for cleanware, is not the focus in this project, it does not matter as long as the TPR for malware is high.

The confusion matrix is depicted in Figure 11.1.

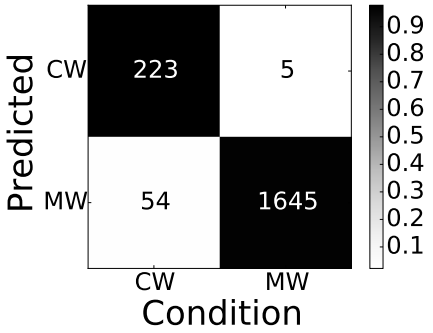


Figure 11.1: Confusion matrix for representation 1.

It is noticed, that the amount of FNs for cleanware is quite high compared to the total number of cleanware used, which also explains the low TPR. In comparison, a very low number of FNs for malware is seen. Compared to the total number of 1,650 malware, only 5 malware samples are not detected. This is also the reason why a high TPR for malware, is noticed.

The classifier for Representation 1, performs well for detecting malware, but has the problem of predicting cleanware as malware. More discussion on the usability of the classifier is done after presenting the

results for Representation 2.

In Figure 11.2, the performance of the classifier is visualized using the a ROC curve for each class. It should be noticed, that the figure shows a zoom of the relevant area of the ROC curves.

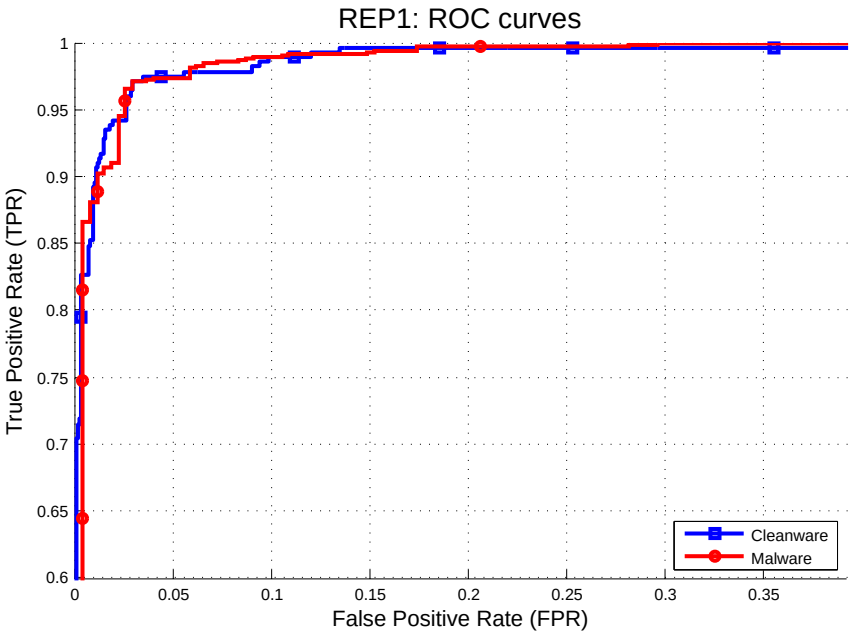


Figure 11.2: ROC curves for Representation 1, zoomed in on the relevant area.

Overall the performance is good for both classes. The reason for having such a good ROC curve for cleanware, is that the FPR is very low, so it is not expected to increase that much when the TPR increases. The opposite is true for malware. Here, the TPR is high, but this is also the case for the FPR. This means, that from the ROC curve, it is seen that with a TPR of 0.975, one could expect a much lower FPR of only 0.05. Note, that from Equation 11.1 (p. 97) an FPR of 0.195 at TPR of 0.997 was seen. Though, a TPR of 0.975, would mean more FNs for malware, thus a worse detection.

In Figure 11.3, the distribution of the different features used during testing, is depicted.

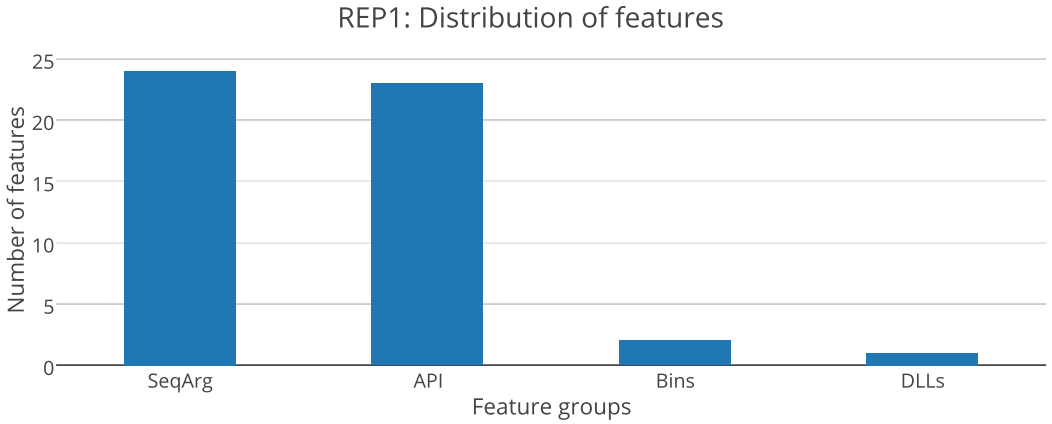


Figure 11.3: Distribution of features for Representation 1.

It is noted, that when switching from 1k to 5k samples, the features selected includes more sequence with arguments. This might be due to the fact, that when more samples exist for training, the classifier can find a more distinctive pattern between malware and cleanware. It is still seen, that the frequency of different APIs has an important role in discriminating between the classes.

From the presented results, it is concluded that for discriminating malware and cleanware, sequence with arguments, can be used. Though, in order to use these features, a higher number of samples are needed. This statement is also based on the training results. Here, it was seen for 10k, 20k and all malware samples, that more than half of the features include sequence with arguments. It is of course known, that adding more cleanware, could change this conclusion, since the sequence with arguments could lose its discriminative power, letting other features to be chosen instead. When examining the frequency of the APIs in WEKA, it seems that most of the APIs chosen by the FS, include a high frequency for malware, whereas cleanware has a lower frequency, close to zero. This can also explain the high amount of cleanware that are falsely classified as malware, as the cleanware, with a high frequency for the APIs compared to other cleanware, makes RF see them as malware.

11.1.2 Representation 2

In Table 11.2 the test results for Representation 2 is shown.

Class	TPR	FPR	PPV	F-Measure	AUC
Cleanware	0.884	0.002	0.984	0.932	0.996
Malware	0.998	0.116	0.981	0.989	0.996
Weighted Avg.	0.981	0.099	0.981	0.981	0.996

Table 11.2: Test results for Representation 2.

Cleanware performs better compared to Representation 1. Here, the TPR for cleanware has increased by 0.08, leaving it at 0.884. This means, that fewer cleanware are detected as malware. Also, the FPR for cleanware has slightly decreased, indicating that more TNs are seen, compared to FPs. This also affects the PPV, making it increase by 0.07, leaving it at 0.984 for cleanware. Since the TPR and PPV both increases for cleanware, the F-measure also increases, from 0.883 to 0.932, which is a high increase. The improvement for cleanware also results in a higher AUC value of 0.996.

Malware also performs slightly better, with a higher TPR of 0.998, and with a decrease in FPR from 0.195 to 0.116. Also, the PPV increases as a result of a lower FPR, here from 0.968 to 0.981, which indicates that the FPs for malware have lowered. This also makes sense, since the TPR for cleanware increased. The F-measure for malware did also increase to 0.989, making Representation 2 give rise to a better classifier for both classes. The AUC is also higher, which conclusively means, that the classifier has a high TPR and low FPR.

Looking at the weighted average, the overall performance for the classifier is good, in relation to the goal of this project. It performs better in both TPR, FPR and PPV compared to Representation 1. That

being said, the FPR is still quite high, and has to be improved in future work. This results in a good F-measure of 0.981, which can be explained by the performance gain for cleanware's TPR. This means, that Representation 2 improves the performance of the classifier, and manages to decrease the number of FPs for Malware. In Figure 11.4 the confusion matrix for Representation 2 can be found.

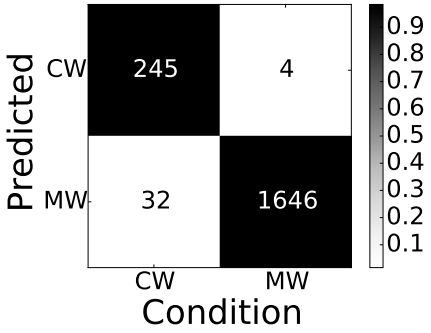


Figure 11.4: Confusion matrix for Representation 1.

It is noticed that the amount of FPs for cleanware, thereby the FNs for malware, have decreased by 1, thus only 4 malware are not detected during the test. Furthermore, the FNs for cleanware, i.e. FPs for Malware, have decrease by a total of 22, which leaves only 32 cleanware falsely classified as malware. This shows, that a reduction of FPs for cleanware, can be done without increasing its FNs. It was not known, which feature representation would perform best, but studying the results, it shows that separating sequence and arguments improves the discrimination. This can be stated since the number of features are the same, but the results have improved.
The performance of the classifier is visualized using the ROC curves in Figure 11.5. It should be noticed, that the figure shows a zoom of the ROC curves.

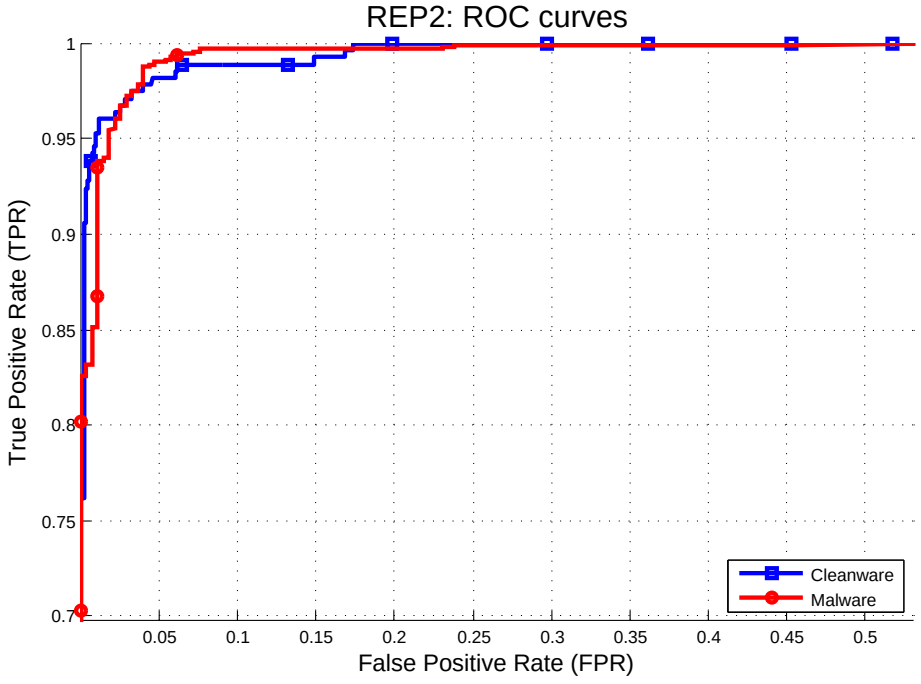


Figure 11.5: ROC curves for Representation 2 zoomed in on the area that is not covered.

Overall the performance is good for both classes, and as emphasized earlier, it performs better than Representation 1. This is expected due to the fact, that both cleanware and malware are performing better. Studying the ROC curve of malware, the graph is steeper, where it this time is expected to have a FPR of 0.05 at a TPR of 0.99. Looking at the graph for cleanware, it is a little steeper, but after a FPR of 0.015 the slope becomes flatter from 0.06 to 0.2, after which it steepens again towards 1 TPR.

In order to see what kind of features, that were selected in the FS, Figure 11.6 shows the distribution.

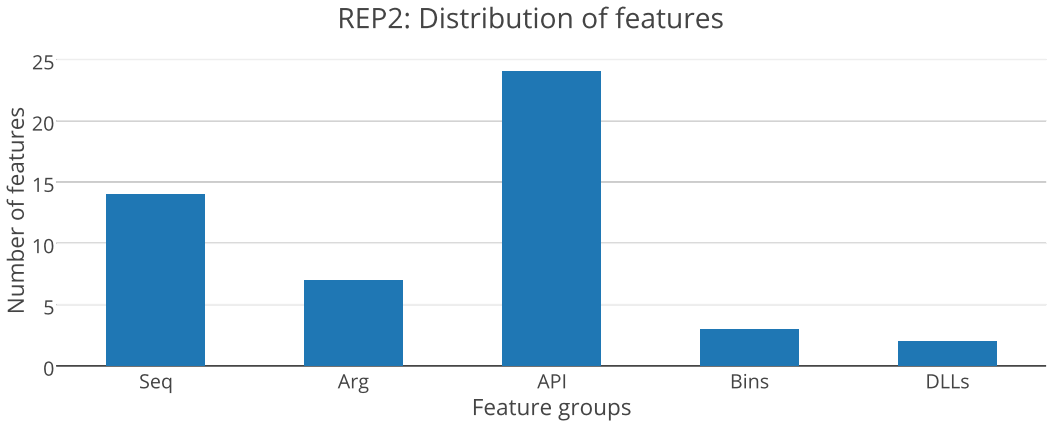


Figure 11.6: Distribution of features for Representation 2.

Separating the sequence and arguments has shown to increase the performance for Representation 2, here noticeably for cleanware. It is seen, that the sequence and arguments are still used a lot. The difference compared to Representation 1, besides from sequence and arguments being separated, is that here the actual argument is present. This might have influenced the FS algorithm into picking more arguments. It seems that combining sequence with arguments, bring out more possible categorical labels, due to the use of nominal features, while separating sequence and arguments gives a smaller number. As for Representation 1, many frequency APIs are chosen, together with several frequency bins and DLLs.

Looking in WEKA, FS uses many APIs that have higher frequency for malware compared to cleanware, as was also seen for Representation 1. It seems, that sequence and argument have a high mixture of both classes, but in a separated manner.

Discussion and Conclusion

The goals for malware detection are to detect as much malware as possible, while at the same time not predicting too many cleanware as malware. This are mostly satisfied for detection, where only 4-5 malware are not detected out of 1,650 malware, which in this study is good performance. Of course, since training included 5k malware samples, the amount of FNs for malware was favored, and gave rise to a higher amount of FPs. So the nonuniform sample set, has the consequence of better detection at the expense of more FPs for malware. Since the priority is to detect malware, the choice of using 5k malware samples is made.

It is known that malware can emulate cleanware behavior by faking API calls, thereby hiding its true behavior, and evading the classifier. However, this was not seen during the training, where only 17 malware were not detected from a total of ≈ 117k malware samples. It is not known, if these malware were not detected because it emulated cleanware behavior.

From analysis in Section 3.1, it was found, that the method used for acquiring analysis data (pre-installing cleanware), could have both a negative or positive effect on the cleanware data. This should be taken into account, when looking at the results, since it could cause an offset in the amount of FPs and FNs.

The results from testing also show, that the methods for type classification, used in the last study, can be used for detection. Furthermore, the frequency APIs and sequence, can actually be used for discrimination between classes. Also FS has shown to improve the results significantly. It can improve the results by removing the features, that are irrelevant and redundant, thus increasing the performance. Additionally, FS can also speed up the algorithm, since less features needs to be taken into account. From FS, it was seen that many of the features with best performance, were the APIs with higher frequency for malware, compared to cleanware.

From the two representations, it is clear that Representation 2 is best. By separating the sequence and arguments, one can get better predictive performance for detection.

It was also found, that sequence first becomes relevant as a feature for detection, when enough samples are used. This was discovered from the training, where switching from 1k to 5k samples, around 20 more sequence features were used. The tendency might not be seen, if more cleanware had been injected.

As a conclusion, Representation 2 is best, with only 4 FNs and 32 FPs for malware. This is based on 277 cleanware and 1,650 malware samples. The TPR for cleanware is 0.884, but with a better PPV than malware, namely 0.984, whereas malware had a good TPR of 0.998 and PPV of 0.981. In total, the project ended with a good classifier, that could be used for malware detection. That being said, future work should focus on decreasing the amount of FPs for malware, even though the FNs are quite low.

11.2 Family Classification

This section includes the test results for Family Classification. The models created in the training phase for both Representation 1 and 2 include 400 and 600 features, respectively chosen by the FS. A discussion and conclusion is presented after the test result for both representations.

11.2.1 Representation 1

The test results for Representation 1 are shown in Table 11.3 (p. 104). All of the classes have relatively uniform sample distribution, where Hupigon has the highest number of instance.

Class	TPR	FPR	PPV	F-Measure	AUC
Small	0.803	0.019	0.910	0.853	0.972
OnlineGames	0.721	0.010	0.927	0.811	0.968
Hupigon	0.964	0.041	0.913	0.938	0.990
Frethog	0.895	0.068	0.719	0.798	0.972
Zlob	0.850	0.033	0.856	0.853	0.979
Weighted Avg.	0.864	0.035	0.872	0.864	0.978

Table 11.3: Test results for Representation 1.

Small overall has a decent performance with a TPR of 0.803 and quite small FPR of 0.019. It has the second lowest number of FPs, which means that it is very few of the other classes that are predicted as Small. Furthermore, it has a PPV above 0.9, which is also the consequence of a low number of FPs. F-measure scored 0.853, where it has to be noted, that this metric takes both the TPR and PPV into account. This shows, that the performance of Small is good and it is also supported by a relatively high AUC-value.

OnLineGames, which is the class with the fewest samples, overall delivers a poor performance compared to the of the classes. This is also noticed when evaluating the TPR, where the score is 0.721. On the other hand, this class has the best score in FPR with 0.010. This is due to the fact, that it has the fewest number of FPs relative to the sum of TNs and FPs. This means, that the class has significantly more FNs compared to FPs, since the TPR is actually the lowest among all the classes. The small amount of FNs is also seen when studying the PPV, which in this case is the highest among the classes, with a score of 0.927. The F-measure is the second worst score with only 0.811, but this performance is still decent. Since this class has the lowest TPR, it also has an affect on the AUC. Here, it also has the worst performance, scoring 0.968.

Hupigon has the highest amount of samples, and is also one of the best performing class. The high amount of samples could explain the good performance, but it is also noticed, that many signature features for this class in particular, was chosen in FS. This is seen by having the highest TPR of 0.964. Though, having a good TPR does not mean it will perform good in all cases. This is also noticed when studying the FPR, which is the second worst. The FPR is also reflected in the PPV, where the performance of 0.913 is mediocre. The F-measure scores 0.938, which again is the highest value together with the AUC of 0.990. A reason for the good performance for Hupigon, could be due to the large sample size. The sample set is larger than the rest of the classes, which are close to uniformly distributed.

Frethog has the second best performance in TPR scoring 0.895. Even though, the TPR is high, it scores the worst FPR of 0.068. This is still a quite low value, but the impact of the relatively high number of FPs compared to the other classes, reduce the performance of the PPV as well. Here, the score is also the worst, with a value of 0.719. Again, due to the high amount of FPs, it leaves the class with the worst performance in F-measure. At last Frethog shares the same value in AUC as Small, here 0.972. It can be concluded based on the F-measure and FPR, that this class performs as poorly as OnLineGames. The AUC of OnLineGames is worse than Frethog's AUC, due to the low TPR, whereas the high FPR for

Frethog does not have as high impact on the AUC in comparison to the TPR.

Zlob has the third best TPR, which means a limited amount of FNs in relation to the amount TPs. It scores in the middle with a value of 0.850, compared to the rest of the classes. The picture is the same for FPR, where it scores 0.033. This leaves a PPV of 0.856, which is the second lowest value for all the classes. This being said, it still has a good performance, which is also supported by the F-measure of 0.853 and the AUC of 0.979.

Viewing the weighted average from the different measures, it is concluded that the classifier performs well. Of course, the case could be different using another dataset or other classes, but these particular features are useful when classifying the chosen families. Studying the details even more, a confusion matrix is depicted in Figure 11.7. This helps to understand what happens between the classes in higher detail, and are used to indicate some relationships or dependencies between the classes. The total number of instances in the testing set is 10,330 and these are divided into all the cells in the confusion matrix, based on the classifiers predictive performance.

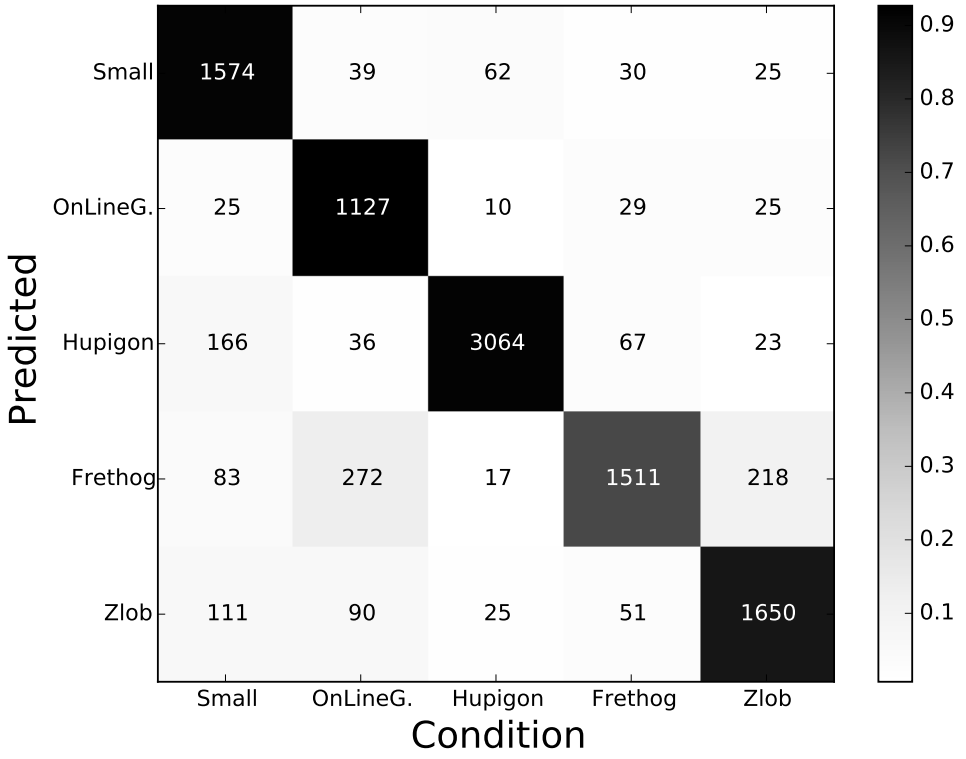


Figure 11.7: Confusion matrix for Representation 1.

Studying Figure 11.7, the overall performance is good, but a significant amount of instances are predicted as Frethog. The remaining classes, have a relatively high number of correctly classified instances.

Digging deeper into the adjacent cells, it is noticed that Small has a high number of FNs (vertical cells) compared to FPs (horizontal cells). Especially, it is observed that Small is predicted as Hupigon and Zlob in the majority of its FNs. This might indicate, that some variants of Small have the same behavior as

Hupigon and Zlob and vice versa. By the descriptions from Microsoft, this observation is also plausible as Small generally consist of Trojans that connects to a remote server. Hupigon is mainly known as RATs, thus the results suggest, that this could be a common trait. Zlob on the other hand is mainly known for hijacking the users' browser, but has also been observed to act as a Trojan downloader, which means it has to connect to a remote server. This might be the reason why many Smalls' are predicted as Zlob. Viewing the distribution of FPs and FNs, a significant amount of OnlineGames samples have been predicted as Frethog. The same picture is seen for Zlob, where 218 of the Zlob samples are predicted wrongly as Frethog. The behavior of this family is known to target password theft from online games, mainly World of Warcraft. Also, OnlineGames is known to log keystrokes, which could explain why many OnlineGames, here 272, are predicted as Frethog. Since World of Warcraft or any other games, were not installed on the VMs, the poor results for Frethog in terms of its FPs, might indicate that not enough information was present in the samples, to create a discrimination between itself and the other classes.

At last, the ROC curves for each individual class are plotted in Figure 11.8.

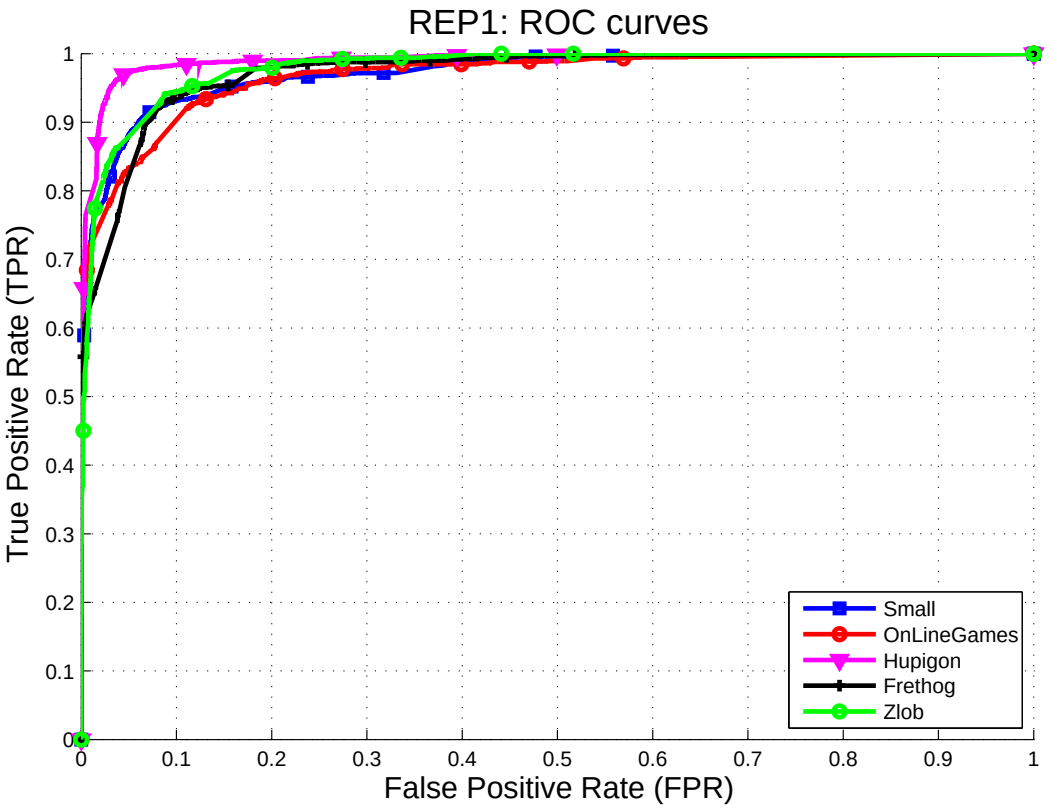


Figure 11.8: ROC curves for Representation 1.

As seen on the figure, Hupigon has the best performance after which the performance of the remaining classes performs similar. Again, this could be due to the higher amount of samples for this class. Studying the graphs a little closer, it can be seen that OnlineGames and Frethog performs worst, which was also supported by the aforementioned discussion. It is observed, that the curves from Zlob and Small are very identical, which means that their predictive performance, i.e. the distribution of FPR and TPR must be very similar.

To conclude, these results show a satisfactory performance, leaving only few FPs and FNs compared to the full number of test instances. The classifier seems to have minor trouble when discriminating OnlineGames and Zlob and instead predicted it as Frethog. This could be the consequence of too few discriminating features for this particular class. The next subsection discusses the test results for Representation 2.

11.2.2 Representation 2

The test results for Representation 2 are shown in Table 11.4.

Class	TPR	FPR	PPV	F-Measure	AUC
Small	0.828	0.031	0.860	0.844	0.970
OnLineGames	0.718	0.010	0.930	0.810	0.967
Hupigon	0.963	0.040	0.914	0.938	0.990
Frethog	0.887	0.067	0.721	0.796	0.971
Zlob	0.816	0.027	0.875	0.845	0.978
Weighted Avg.	0.860	0.036	0.867	0.860	0.977

Table 11.4: Test results for Representation 2.

As a general comment on these results it is seen, that it performs slightly worse than Representation 1. **Small** scores a TPR of 0.828, which is slightly better than for Representation 1. On the other hand, FPR is worse with a minor increase of 0.031. This is still a relatively good value, but a lower value is desirable. The impact of the increased FPR is the decrease in PPV, which now scores below 90 %. This is due to the increased amount of FPs in the classification. Because PPV defines the F-measure, a decrease to 0.844 is also seen here. Nevertheless, the overall performance is still decent. It scores an AUC of 0.970. **OnLineGames**, which in this test also has the fewest number of samples, has the lowest TPR in the classification, scoring only 0.718. This was also the case in the aforementioned section. Though, it is noticed that the value is lower here. The FPR is actually the best again, scoring the same value as before. The FPs and TPs are reflected in the PPV, but since the ratio is different the PPV is higher. Moving on to the F-measure, it has a lower value compared to Representation 1, which is also the case for the AUC. Overall OnLineGames has the worst performance when focusing on the TPR, but this is balanced by the good FPR. It can also be discussed that the low TPR is the consequence of having the fewest instances when the model was trained.

Hupigon more or less has the same performance as for Representation 1. It has the highest amount of instances among the classes. With the same reasoning as before, the high amount of samples could explain the good performance. Additionally, it is noticed that many signature features for this class were chosen in the FS. This is verified by the TPR of 0.964. Though, it is noticed that it has the second worse FPR, here 0.041, but this value is still relatively low. The high amount of TPs and FPs result in a PPV of 0.913. This value is the second best among all the classes, but worse due to the large amount of FPs. Moving on to the F-measure and AUC, Hupigon scores 0.938 and 0.990 respectively, which is the same as for Representation 1. Overall, Hupigon has a very satisfactory performance.

Frethog has the second best performance in TPR scoring 0.887, but in contrast has the worst score in FPR of 0.067. The high amount of FPs result in the worst performance in PPV as well, here only scoring 0.721. This influences the F-measure, where it also has the worst score of 0.796. Nevertheless, due to the high TPR, the AUC scores 0.971, which is the third best among the classes.

Zlob, again has a mediocre performance in TPR scoring a value of 0.816. This is also the case for the FPR of 0.027, which is better than for Representation 1. Looking at the PPV, it scores 0.875, which is a decent performance. This is also better compared to the first representation and is also reflected in the F-measure and AUC, where it scores 0.845 and 0.978 respectively. Overall the performance of this class is close to the same as in Representation 1, but last mentioned scores better in the TPR but lacks performance in FPR.
At last, viewing the weighted average, the performance in general is very satisfactory. Comparing the findings with the results for Representation 1, it is observed, that it performs slightly worse. Though, the differences are quite small, ≈ 0.005 for both TPR and PPV. Even a small difference in the predictive performance is significant, when testing a large amout of samples.

Studying the details even more among the individual classes, the confusion matrix is depicted in Figure 11.9.

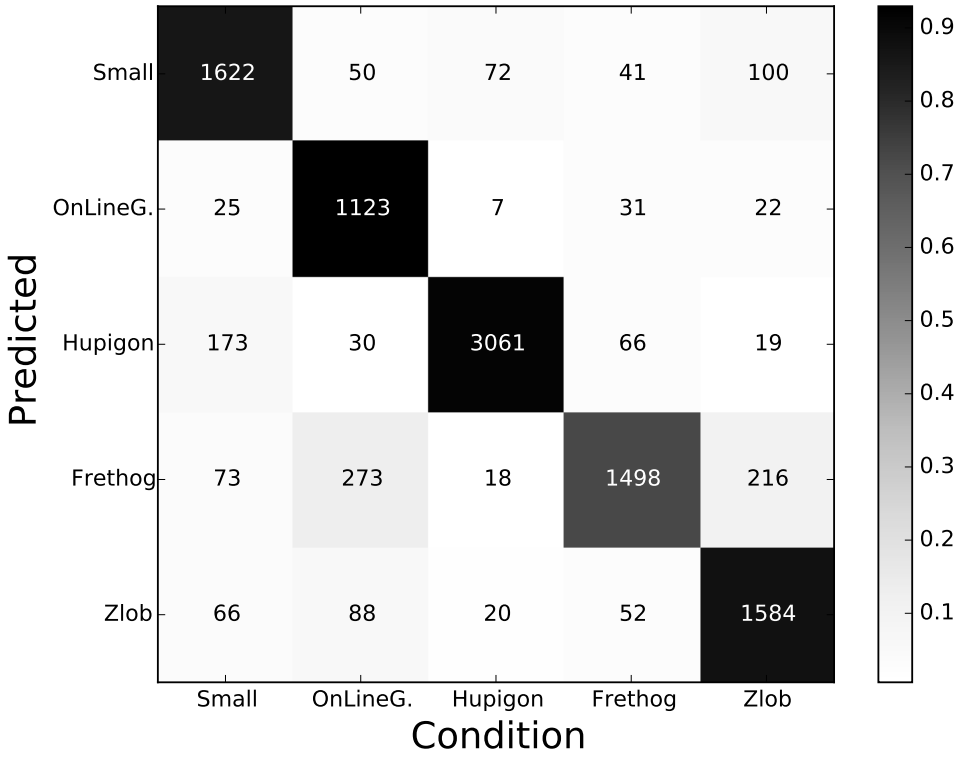


Figure 11.9: Confusion matrix for Representation 2.

The confusion matrix generally is quite similar to Figure 11.8. It is noticed that Small has a larger amount of FPs, which was also confirmed by the results shown in Table 11.4. Especially, Zlob is predicted 100

times as Small. With the same reasoning as before, it can be assumed, that this is because both Small and Zlob are known to download additional unwanted software, where they might connect to a remote server. The same can be said for Hupigon, which also has a significant amount of FPs, which are predicted as Small. Hupigon is a very large family known to be RATs. Studying the FPs and FNs for Frethog, it had one of the poorest performances in the last test, which is also confirmed for this representation. The similarity here might be because the same information is contained in both representations. A significant amount of OnlineGames and Zlobs are predicted as Frethog. This leaves Frethog with the highest amount of FPs compared to the rest of the classes. Using the reasoning from before, the behavior of Frethog is known to be very different from the remaining classes with the exception of OnlineGames, which might the reason it has trouble discriminating the two. On the other hand, no online games were installed on the VMs, which means that a lot of potential behavior was not executed, leaving the classes less defined. In other words, it is believed that because the models of the other classes are better defined, it will tend to favor Frethog when it evaluates a sample, which is hard to discriminate.

Even though, Frethog had a lot of FPs compares to the other classes, the classifier performs satisfactory. This is also confirmed, when studying the individual ROC curves in Figure 11.10.

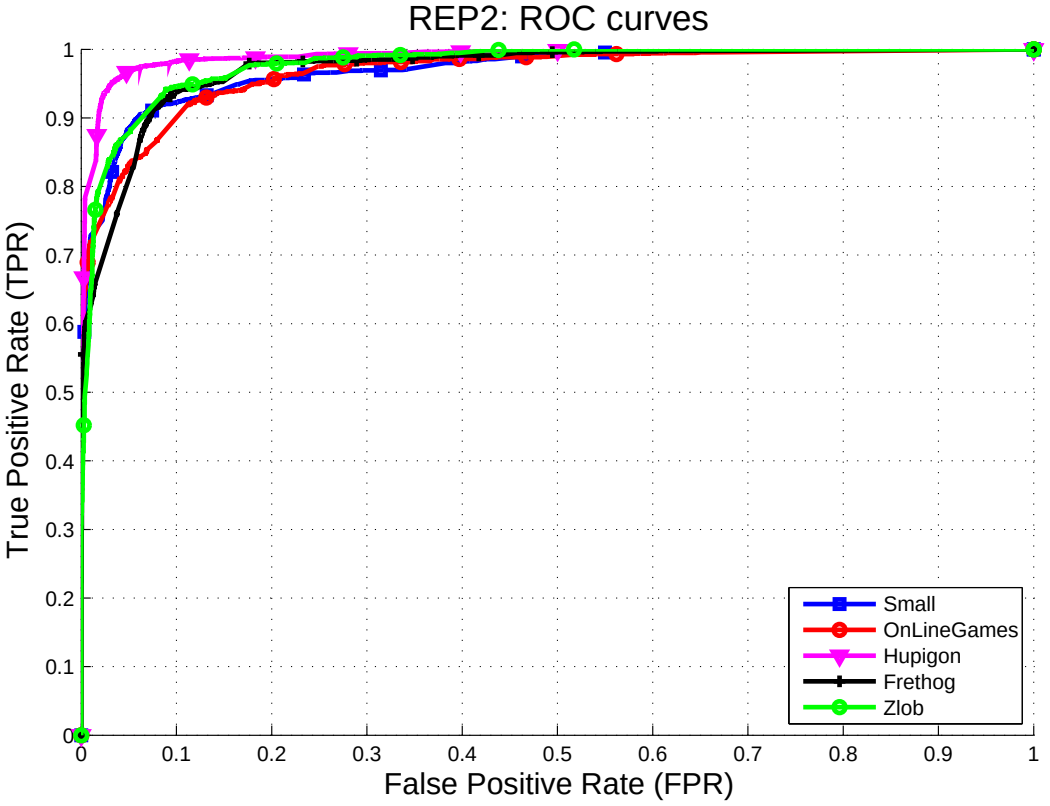


Figure 11.10: ROC curves for Representation 2.

The graphs look identical to Figure 11.8 (p. 106), but analyzing the interval from 0-0.2 FPR, minor differences are noticed. Small has slightly decreased its performance as the curve is less steep than for Representation 1. It is the same case for OnlineGames, Frethog and Zlob, which all three have decreased 0.001 in AUC. Only Hupigon, which remains the best class, has the same behavior. The performance for

Representation 2 are satisfactory.

Discussion and Conclusion

One of the goals in this study was to achieve better uniformity in the dataset, compared to the last study. This was achieved for four out of five classes, which all had between 4k-6k. Hupigon differed from the rest with close to 10k samples, which might also explain why this class has the best performance. That being said, the performance of the rest of the classes were still satisfying.

By studying the results even more, and realizing some of the individual and seemingly correlated behaviors, it was noticed that three of the classes, namely Small, Hupigon and Zlob possibly had some behavior in common. It is assumed that the similarity between the classes, lie in the characteristic of accessing a remote server. The same can be said between Frethog and OnlineGames, which also have similar behavior, i.e. stealing information and key-logging. A solution to this, could be to find different features known to discriminate the families even more. This could for instance be specific signatures, which can be found by analyzing more AV-reports or studying the behavior manually.

The overall assessment of the family classification is very satisfactory, when looking at the results. It was noticed that the differences in performance were not significantly different, but both representations performed better than expected, with weighted average TPR above 0.85, FPR below 0.04, PPV above 0.86, F-measure above 0.85 and AUC above 0.97. Conclusively, the family classification shows that Representation 1 is the best representation when classifying malware families. Note, here that in the training Representation 2 actually performed best, but it is the testing that matters.

Part V

Conclusion

INTENTIONALLY LEFT BLANK

CHAPTER 12
CONCLUSION

This project addressed the problem increased amount of new malware, which emerge every day. In the last couple of years the amount of malware has tripled, and approximately 390,000 new malware are registered each day. A high amount of new malware, makes analyzing them manually difficult, leaving automated tools for malware analysis a necessity. Therefore the following problem statement was derived for this project:

How to perform Malware Detection and Family Classification using Machine Learning?

This project is a continuation of a 9th semester project, where malware were classified into types. Since the definitions of types are ambiguous, leading to class overlaps, this study has utilized malware families, which are believed to provide more discriminative behavior. Therefore, this study has addressed family classification and in addition to this, extended the system to include malware detection. The newly proposed detection approach is intended for private users and potential firms, while the family classification is intended for AV-vendors, to overcome the problem of a the exponential growth of new malware.

The project was realized using a modified version of Cuckoo Sandbox to perform dynamic analysis, where specific features were extracted for malware. These features were used in supervised ML to perform malware detection, in terms of a binary classifier and family classification. Three different representation techniques were combined, namely sequence, frequency and binary. The features for the sequence were represented in two different versions: Representation 1 included the APIs combined with its arguments, resulting in one feature, and Representation 2, where the API and argument act as two separate features. These representations were combined with frequency representations. Frequency was used for APIs, API bins and DLLs. The binary representation was specifically used for family classification to represent signatures belonging to specific families.

To improve the detection and family classification, FS was used to remove redundant and irrelevant features with the goal of increasing the predictive performance of the ML algorithm. Based on experience from the last study, RF is used and provided satisfactory results. The results for detection using Representation 1, scored a weighted average of 0.969 in TPR and 0.970 in PPV. Representation 2, contributed to a TPR of 0.981 and PPV of 0.981. This is a good performance for malware detection, and it is also noticed, that separating the APIs from their arguments, contributed to the best performance. It was found that the performance could be improved by solving the issues where cleanware is falsely classified as malware. In contrast, the classifier performed well in terms of FNs for malware, where only 4 FNs were present out of 1,650 samples. To conclude, by eliminating the number of FPs for malware, a detection using dynamic analysis, is feasible to be used for private users.

The results for family classification for Representation 1, scored a weighted average TPR of 0.865, a PPV of 0.872 and an AUC of 0.978. For Representation 2, the results were slightly worse, scoring a weighted

average TPR of 0.860, a PPV of 0.867 and an AUC of 0.977. It is noticed, that Representation 1 is better for family classification, whereas Representation 2 is better for detection. For Representation 1, the predictive performance is high for most of the family classes, whereas OnlineGames and Frethog, seems to conflict with each other, as both classes potentially exhibit similar behavior. To conclude, if these two classes are improved, classification using families, are believed to be feasible and useful for AV-vendors.

Since execution of cleanware programs in Cuckoo was not straight forward, future work needs to facilitate a way of executing legit software already installed on the machine. This could for instance be done by providing Cuckoo with a path to programs to analyze in the VMs. This study executed shortcuts pointing to programs, which introduced data, that had to be deleted afterwards. Furthermore, it is known that MongoDB has a limit of 16 MB when importing data, which meant that reports exceeding this limit, could not be imported. Solving this problem, it is possible to include the installation processes of cleanware programs and thereby potentially increase the class discrimination. Additionally, it could be used to detect malware hiding in legit software, where the malware is released during the installation. The presented malware detection and family classification approaches could be improved by incorporating detection of novel malware or new families. This allows it to be used as a pre-filtering system for AV-vendors. Here, it is imagined, that if the classifier would be able to identify all families from a AV-vendor and reject malware, where the uncertainty of family identification is below a specific threshold. For the overall system, further research should be put into optimizing the analysis time. Reducing the analysis time without lowering the predictive performance, would result in a very efficient detection and classification system, that can cope with the exponential growth of malware.

Part VI

References

INTENTIONALLY LEFT BLANK

CHAPTER 13

LIST OF REFERENCES

[Alazab et al., 2010] Alazab, M., Venkataraman, S., and Watters, P. (2010). Towards understanding malware behaviour by the extraction of api calls. http://ieeexplore.ieee.org/.

[AV-test, 2015] AV-test (2015). New malware. http://www.av-test.org/en/statistics/malware/.

[Beal, 2015] Beal, V. (2015). Api - application program interface. http://www.webopedia.com/TERM/A/API.html.

[Blasco, 2009] Blasco, J. (2009). Malware: Exploring mutex objects. https://www.alienvault.com/open-threat-exchange/blog/malware-exploring-mutex-objects.

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

[CARO, 2014] CARO (2014). What is the issue? http://www.caro.org/naming/scheme.html.

[Codenomicon, 2014] Codenomicon (2014). Heartbleed. http://heartbleed.com/.

[de Freitas, 2013] de Freitas, N. (2013). Machine learning - random forest. https://www.youtube.com/watch?v=3kYujfDgmNk.

[Deng and Runger, 2012] Deng, H. and Runger, G. (2012). Feature selection via regularized trees. *Neural Networks (IJCNN)*.

[ENISA, 2015] ENISA (2015). 1st pan-european conference on cyber security and privacy challenges for law enforcement. https://www.enisa.europa.eu/media/news-items/successful-conclusion-for-the-joint-conference-by-enisa-and-the-heraklion-chamber-of-commerce-and-industry-on-cyber-security.

[Evans, 2011] Evans, K. (2011). What is dll hijacking? http://resources.infosecinstitute.com/dll-hijacking/.

[F-Secure, 2015] F-Secure (2015). F-secure malware knowledge base. https://www.f-secure.com/v-descs/backdoor_w32_hupigon.shtml.

[FBI, 2015] FBI (2015). Ransomware on the rise - fbi and partners working to combat this cyber threat. http://www.fbi.gov/news/stories/2015/january/ransomware-on-the-rise/ransomware-on-the-rise.

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

[Fisher, 2015] Fisher, T. (2015). Windows registry. http://pcsupport.about.com/od/termsr/p/registrywindows.htm.

[Franklin and Coustan, 2015] Franklin, C. and Coustan, D. (2015). How operating systems work. http://computer.howstuffworks.com/operating-system9.htm.

[Fukushima et al., 2010] Fukushima, Y., Sakai, A., Hori, Y., and Sakurai, K. (2010). A behavior based malware detection scheme for avoiding false positive. *Secure Network Protocols (NPSec), 2010 6th IEEE Workshop on*, pages 79–84.

[Guyon et al., 2006] Guyon, I., Gunn, S., Nikravesh, M., and Zadeh, L. A. (2006). *Feature Extraction: Foundations and Applications*. Springer.

[Guyon et al., 2010] Guyon, I., Saffari, A., Dror, G., and Cawley, G. (2010). Model selection: Boyond the bayesian/frequentist divide. *The Journal of Machine Learning Research*, 11:61–87.

[Gönen, 2007] Gönen, M. (2007). *Analyzing receiver operating characteristic curves with SAS*. SAS Institute Inc.

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

[Hastie et al., 2009] Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning - Data Mining, Inference, and Prediction*. Springer.

[Hughes, 1968] Hughes, G. F. (1968). On the mean accuracy of statistical pattern recognizers. *IEEE Transactions on Information Theory*.

[Jiang et al., 2011] Jiang, Q., Zhao, X., and Huang, K. (2011). A feature selection method for malware detection. *Information and Automation (ICIA)*, pages 890–895.

[Kasama et al., 2012] Kasama, T., Yoshioka, K., Inoue, D., and Matsumoto, T. (2012). Malware detection method by catching their random behavior in multiple executions. *Applications and the Internet (SAINT), 2012 IEEE/IPSJ 12th International Symposium on*, pages 262–266.

[Konstan, 2014] Konstan, J. (2014). Basic decision support metrics. https://class.coursera.org/recsys-001/lecture/139.

[Krzanowski and Hand, 2009] Krzanowski, W. J. and Hand, D. J. (2009). *ROC Curves for Continous Data*. CRC Press.

[Kumar, 2014] Kumar, V. (2014). Feature selection: A literature review. *Smart Computing Review*, 4(3).

[Larsen et al., 2014] Larsen, T. M. T., Hansen, S. S., Pirscoveanu, R. S., and Czech, A. (2014). Analysis of malware behavior: Type classification uusing machine learning. Technical report, Aalborg University.

[Liu et al., 2013] Liu, J., Song, J., Miao, Q., and Cao, Y. (2013). Fenoc: An ensemble one-class learning framework for malware detection. *Computational Intelligence and Security (CIS), 2013 9th International Conference on*, pages 523–527.

[Lupo PenSuite Collections, 2015] Lupo PenSuite Collections (2015). Lupo pensuite collections. http://www.lupopensuite.com/collection.htm.

[Malwarebytes, 2013] Malwarebytes (2013). Obfuscation: Malware's best friend. https://blog.malwarebytes.org/intelligence/2013/03/obfuscation-malwares-best-friend/.

[Manning, 2014] Manning, C. (2014). Precision, recall and the f measure. https://class.coursera.org/nlp/lecture/142.

[Markel and Bilzor, 2014] Markel, Z. and Bilzor, M. (2014). Building a machine learning classifier for malware detection. *Anti-malware Testing Research (WATeR), 2014 Second Workshop on*, pages 1–4.

[Microsoft, 2014] Microsoft (2014). Microsoft threat encyclopedia. http://www.microsoft.com/security/pc-security/malware-families.aspx.

[Microsoft, 2015a] Microsoft (2015a). Api index. https://msdn.microsoft.com/en-us/library/windows/desktop/hh920508(vs.85).aspx.

[Microsoft, 2015b] Microsoft (2015b). Glossary. http://www.microsoft.com/security/portal/mmpc/shared/glossary.aspx.

[Microsoft, 2015c] Microsoft (2015c). Mutex objects. https://msdn.microsoft.com/en-us/library/windows/desktop/ms684266(v=vs.85).aspx.

[Microsoft, 2015d] Microsoft (2015d). Naming malware. http://www.microsoft.com/security/portal/mmpc/shared/malwarenaming.aspx.

[Microsoft, 2015e] Microsoft (2015e). Ntopendirectoryobject function. https://msdn.microsoft.com/en-us/library/bb470234(v=VS.85).aspx.

[Microsoft, 2015f] Microsoft (2015f). Object directories. https://msdn.microsoft.com/en-us/library/windows/hardware/ff557755(v=vs.85).aspx.

[Microsoft, 2015g] Microsoft (2015g). What is a dll? http://support.microsoft.com/en-us/kb/815065.

[Microsoft, 2015h] Microsoft (2015h). What is the registry? http://windows.microsoft.com/en-gb/windows-vista/what-is-the-registry.

[Ng, 2014] Ng, A. (2014). Machine learning. https://class.coursera.org/ml-005/lecture/preview.

[Ninite, 2015] Ninite (2015). Ninite. https://ninite.com/.

[Python Documentation, 2015] Python Documentation (2015). pickle - python object serialization. https://docs.python.org/2/library/pickle.html.

[Python Software Foundation, 2012] Python Software Foundation (2012). arff - python package for reading and writing weka arff files. https://code.google.com/p/arff/.

[Saini et al., 2014] Saini, A., Gandotra, E., Bansal, D., and Sofat, S. (2014). Classification of pe files using static analysis. *Proceedings of the 7th International Conference on Security of Information and Networks*.

[Salehi et al., 2012] Salehi, Z., Ghiasi, M., and Sami, A. (2012). A miner for malware detection based on api function calls and their arguments. *Artificial Intelligence and Signal Processing (AISP), 2012 16th CSI International Symposium on*, pages 563–568.

[Santos et al., 2011] Santos, I., Brezo, F., Sanz, B., Laorden, C., and Bringas, P. (2011). Using opcode sequences in single-class learning to detect unknown malware. *Information Security, IET*, 5(4):220–227.

[Santros, 2014] Santros, R. (2014). Poweliks: Malware hides in windows registry. http://blog.trendmicro.com/trendlabs-security-intelligence/poweliks-malware-hides-in-windows-registry/.

[Sayad, 2015] Sayad, S. (2015). Zeror. http://www.saedsayad.com/zeror.htm.

[Shabtai et al., 2011] Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y. (2011). "andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*.

[Shahzad and Lavesson, 2011] Shahzad, R. and Lavesson, N. (2011). Detecting scareware by mining variable length instruction sequences. *Information Security South Africa (ISSA), 2011*, pages 1–8.

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

[Symantec, 2015] Symantec (2015). Symantec security response. http://www.symantec.com/security_response/writeup.jsp?docid=2005-102509-1758-99&tabid=2.

[Tian et al., 2010] Tian, R., Islam, R., Batten, L., and Versteeg, S. (2010). Differentiating malware from cleanware using behavioural analysis. *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, 5(5):23–30.

[Trendmicro, 2015] Trendmicro (2015). Trendmicro threat encyclopedia. http://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/.

[Uppal et al., 2014] Uppal, D., Sinha, R., Mehra, V., and Jain, V. (2014). Malware detection and classification based on extraction of api sequences. *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*, pages 2337–2342.

[Vinod et al., 2012] Vinod, P., Laxmi, V., and Gaur, M. (2012). Reform: Relevant features for malware analysis. *Advanced Information Networking and Applications Workshops (WAINA), 2012 26th International Conference on*, pages 738–744.

[VirusShare, 2014] VirusShare (2014). Virusshare 80k samples. http://tracker.virusshare.com:6969/torrents/VirusShare_00139.zip.torrent?CF174A50911F0AD662849A121A1D0D7C18140C8D.

[VirusTotal, 2014] VirusTotal (2014). Virustotal : Free online virus, malware and url scanner. https://www.virustotal.com/en/documentation/.

[VX Heavens, 2010] VX Heavens (2010). Vx heavens snapshot (2010-05-18). https://archive.org/details/vxheavens-2010-05-18.

[WEKA, 2015] WEKA (2015). Weka documentation. Waikato Environment for Knowledge Analysis.

[Wiki-security, 2015] Wiki-security (2015). Method of infection. http://www.wiki-security.com/wiki/Parasite_Category/Adware/.

[Wikipedia, 2014a] Wikipedia (2014a). Confusion matrix. http://en.wikipedia.org/wiki/Confusion_matrix.

[Wikipedia, 2014b] Wikipedia (2014b). F1 score. http://en.wikipedia.org/wiki/F1_score.

[Wikipedia, 2014c] Wikipedia (2014c). Receiver operating characteristic. http://en.wikipedia.org/wiki/Receiver_operating_characteristic.

[Wikipedia, 2014d] Wikipedia (2014d). Sensitivity and specificity. http://en.wikipedia.org/wiki/Sensitivity_and_specificity.

[Wikipedia, 2015a] Wikipedia (2015a). Application programming interface. http://en.wikipedia.org/wiki/Application_programming_interface.

[Wikipedia, 2015b] Wikipedia (2015b). Data pre-processing. http://en.wikipedia.org/wiki/Data_pre-processing.

[Wikipedia, 2015c] Wikipedia (2015c). Feature selection. http://en.wikipedia.org/wiki/Feature_selection.

[Wikipedia, 2015d] Wikipedia (2015d). Feature vector. http://en.wikipedia.org/wiki/Feature_vector.

[Wikipedia, 2015e] Wikipedia (2015e). Kernel method. http://en.wikipedia.org/wiki/Kernel_method.

[Wu et al., 2011] Wu, L., Ping, R., Ke, L., and Hai-xin, D. (2011). Behavior-based malware analysis and detection. *2011 First International Workshop on Complexity and Data Mining*.

[Zolotukhin and Hamalainen, 2014] Zolotukhin, M. and Hamalainen, T. (2014). Detection of zero-day malware based on the analysis of opcode sequences. *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*, pages 386–391.

INTENTIONALLY LEFT BLANK

Part VII

Appendices

INTENTIONALLY LEFT BLANK

APPENDIX A
LIST OF SIGNATURES

This section/appendix will include a complete list of signatures for each of the 5 families, which will be the input values for the API functions. These will in this project be used as family signatures.

Hupigon

1. 6600.org
2. BEI_ ZHU
3. GrayPigeon
4. Hacker.com.cn.exe
5. huaihuaitudou
6. Rejoice2007
7. calc.exe
8. mmc.exe
9. mspaint.exe
10. mstsc.exe
11. osk.exe
12. sndrec.exe
13. sndvol32.exe
14. winchat.exe
15. .com.cn_ MUTEX
17. HKLM\System\CurrentControlSet\Services\system32 ImagePath = C:\WINDOWS\Hacker.com.cn.exe
18. HKLM\System\CurrentControlSet\Services\system32\Security

Small

1. HKEY_ LOCAL_ MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
2. HKEY_ LOCAL_ MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\ OvMon
3. HKEY_ LOCAL_ MACHINE\System\ControlSet001\Services\ Windows Overlay Components
4. HKEY_ LOCAL_ MACHINE\System\CurrentControlSet\Services\Windows Overlay Components

Zlob

1. messenger\msmsgs.exe
2. tgbrfv_ .exe
3. winlogon.exe
4. services.exe
5. spoolsv.exe
6. msmsgs.exe
7. vmsrvc.exe
8. spool\prtprocs\w32x86\
9. ernel32.dll
10. HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\policies\explorer\run
11. HKLM\SYSTEM\CurrentControlSet\Services\pnpsvc
12. HKLM\SYSTEM\CurrentControlSet\Services\EventLog\Application\pnpsvc
13. HKLM\SYSTEM\CurrentControlSet\Control\SafeBoot\Minimal\pnpsvc
14. spool\PRTPROCS\W32X86\000029cc.tmp
15. HKEY_ LOCAL_ MACHINE\SYSTEM\CurrentControlSet\Enum\Root\ LEGACY_ MSWU-B04BA347
16. HKEY_ LOCAL_ MACHINE\SYSTEM\CurrentControlSet\Enum\Root\ LEGACY_ MSWU-B04BA347\0000
17. HKEY_ LOCAL_ MACHINE\SYSTEM\CurrentControlSet\Services\MSWU-b04ba347

18. HKEY_ LOCAL_ MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\policies\system EnableLUA = "0"
19. HKEY_ LOCAL_ MACHINE\SOFTWARE\Microsoft\Security Center UacDisableNotify = "1"
20. HKEY_ LOCAL_ MACHINE\SYSTEM\CurrentControlSet\Services\SharedAccess\Parameters\FirewallPolicy\DomainProfile\AuthorizedApplications\List

Frethog

1. autorun.inf
2. rttrwq.exe
3. 0s63el.exe
4. mkfght0.dll

1. HKEY_ CURRENT_ USER\SoftWare\Microsoft\Windows\CurrentVersion\Run
2. HKEY_ LOCAL_ MACHINE\SoftWare\Microsoft\Windows\CurrentVersion\Run
3. HKEY_ CURRENT_ USER\Software\Microsoft\Windows\CurrentVersion\Run ertyuop
4. HKEY_ CURRENT_ USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced ShowSuperHidden = "0"
5. HKEY_ CURRENT_ USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced Hidden = "2"
6. HKEY_ LOCAL_ MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced\Folder\Hidden\SHOWALL CheckedValue = "0"
7. HKEY_ CURRENT_ USER\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer NoDriveTypeAutoRun = "91"

OnLineGames

1. HKEY_ LOCAL_ MACHINE\SOFTWARE\Classes\CLSID*\InprocServer32
2. HKEY_ LOCAL_ MACHINE\Software\Microsoft\Windows\CurrentVersion\Explorer\ShellExecuteHooks
3. HKEY_ LOCAL_ MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Browser Helper Objects

APPENDIX B
ANALYSIS SYSTEM SETUP

The network details of the analysis system setup can be seen in Figure B.1. The setup consist of a Server, which contains three programs, namely INetSim, Cuckoo Sandbox and MongoDB. Each of these will be explained below in its own section, with reference to Figure B.1.
It should be noted that the setup shown in the Figure B.1, is only used for malware analysis. The setup for cleanware analysis, follows the same structure regarding network details as for the malware analysis setup. The only difference is, that all VMs for the cleanware analysis setup, are located on the same machine, as INetSim and Cuckoo Sandbox.

B.1 Cuckoo Sandbox

The Cuckoo Sandbox configuration, was done in cuckoo.conf file, which can be found on the CD in `\Programs\Cuckoo\CuckooMalwareSetup.zip`. The IP and port of the resultserver are configured to the following:

```
ip = 192.168.1.100
port = 2042
```

The IP will be the standard gateway for all VMs, whereas above port needs to be different from the ports used in common network protocols. This is needed, since it should not conflict with any services used by INetSim, which emulates different protocols on the same IP and listens to specific ports.
For Cuckoo to keep track and log the analysis process for each sample, a database is used. This database will also be used in the study to find the samples that were successfully analyzed, such that only these samples can be imported to MongoDB. A database named malware was made in MySQL, with a user name called "malware" and password "123". In the configuration file, the following line was inserted, such that Cuckoo could use the database:

```
connection = mysql://malware:123@localhost/malware
```

The VMs connected to Cuckoo Sandbox, need the following settings in Windows for IPv4, to be able to connect to Cuckoo and INetSim.

```
SubnetMask:       255.255.255.0
Standardgateway:  192.168.1.100
DNS-server:       192.168.1.100
```

As it can be seen, Cuckoo and INetSim, which also include the DNS-server, are located on the same IP address.
Moving on to the network setup of VMs, it should be noted that each VM controller has an ID number, which will denote its host address in the IP address. So VM Controller 1, has IP 192.168.1.1, whereas VM controller 2, is assigned IP 192.168.1.2. This can also be seen in Figure B.1. The host address for the IP of each VM, depends on the VM's ID, and ID of the VM controller. So the host address of VM

1 on VM Controller 1, is 192.168.1.11, whereas VM1 on VM Controller 2, is 192.168.1.21. This can also be seen in Figure B.1. The configuration of IPs for the VMs were done in virtualbox.conf, where all IP settings for each VM was inserted.

B.2 INetSim

INetSim is configured such that it uses the same IP as Cuckoo Sandbox. Also the DNS-server is located on this address. The configuration file can be found on the CD in folder `\Programs\INetSim`. Below the following changes made in the configuration file for INetSim can be found:

```
1 service\_bind\_address     192.168.1.100
2 dns\_default\_ip           192.168.1.100
3 dns\_static                dns.msftncsi.com 131.107.255.255
4 service\_max\_childs       30
```

It is not a problem that Cuckoo and INetSim are located on the same IP, since INetSim, only responds to known protocols. These protocols do not use port 2042, namely the only port Cuckoo uses. Therefore, this configuration should not make any conflicts. The configuration done in line 3, is to ensure that this address resolves the correct IP. This is needed to make Windows believe that it is connected to the Internet, and thereby turning its NCSI icon on.
The last change to the configuration file, was to enable INetSim to handle 30 parallel connections, named childs in INetSim.

B.3 MongoDB

The mongoDB is also located on the Server, it connects to IP 172.26.12.229 with the default port 27017.

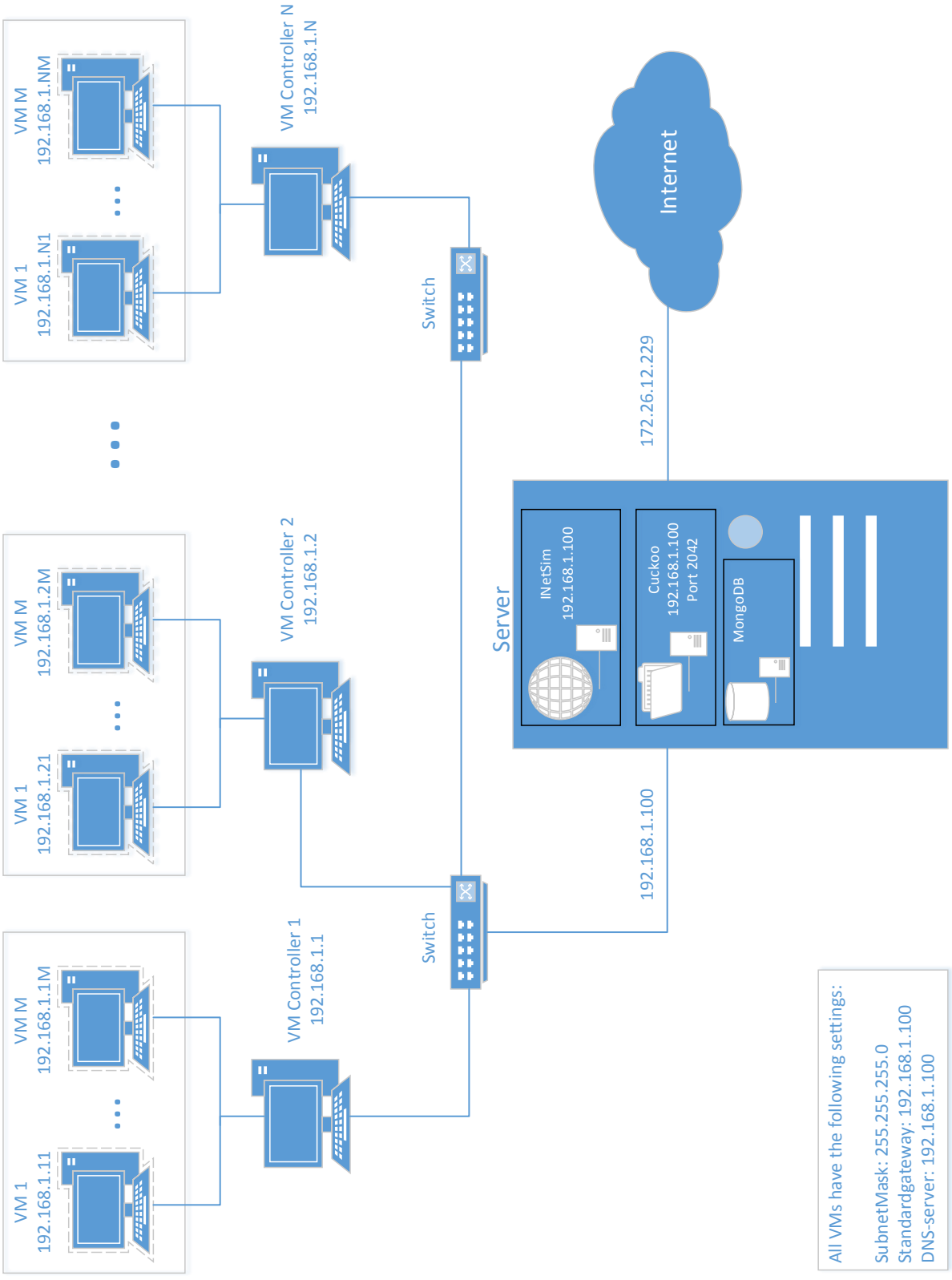


Figure B.1: Network settings for the analysis setup.

APPENDIX C

API NUMBERING

1. NtOpenSection
2. NtDeleteValueKey
3. WSARecv
4. getaddrinfo
5. connect
6. NtResumeThread
7. NtMakeTemporaryObject
8. RegEnumKeyExA
9. NtReadVirtualMemory
10. send
11. NtDelayExecution
12. ShellExecuteExW
13. FindWindowExA
14. RegOpenKeyExA
15. ZwMapViewOfSection
16. GetCursorPos
17. __anomaly__
18. RegEnumValueW
19. NtCreateFile
20. TransmitFile
21. RegEnumKeyExW
22. NtCreateProcess
23. recvfrom
24. sendto
25. NtSuspendThread
26. NtQueryInformationFile
27. RegCreateKeyExW
28. DeviceIoControl
29. WSASocketA
30. CopyFileW
31. OpenServiceA
32. WriteProcessMemory
33. WSARecvFrom
34. NtSetContextThread
35. RegQueryValueExA
36. RemoveDirectoryW
37. OpenServiceW
38. NtSetValueKey
39. ExitProcess
40. WSASocketW
41. RegCreateKeyExA
42. RemoveDirectoryA
43. RegQueryValueExW
44. WSASend
45. NtDeviceIoControlFile
46. NtReadFile
47. RegEnumValueA
48. NtWriteFile
49. LdrGetDllHandle
50. NtWriteVirtualMemory
51. NtEnumerateValueKey
52. CreateDirectoryExW
53. CreateThread
54. NtLoadKey
55. ExitThread
56. recv
57. NtProtectVirtualMemory
58. setsockopt
59. RegDeleteValueW
60. NtGetContextThread
61. RegSetValueExW
62. WriteConsoleA
63. LdrGetProcedureAddress
64. NtOpenThread
65. CreateProcessInternalW
66. RegSetValueExA
67. NtSaveKeyEx
68. NtEnumerateKey
69. NtOpenDirectoryObject
70. LdrLoadDll
71. NtCreateUserProcess
72. URLDownloadToFileW
73. WriteConsoleW
74. RegCloseKey
75. NtSetInformationFile
76. NtCreateKey
77. MoveFileWithProgressW
78. ioctlsocket
79. WSAStartup
80. NtTerminateThread
81. NtTerminateProcess
82. RegOpenKeyExW
83. shutdown
84. DeleteService
85. select
86. NtQueryKey
87. CreateRemoteThread
88. GetSystemMetrics
89. NtCreateThreadEx
90. CreateServiceA
91. RegQueryInfoKeyA
92. InternetCloseHandle
93. system
94. DeleteFileA
95. ReadProcessMemory
96. CreateDirectoryW
97. DeleteFileW
98. NtDeleteFile
99. CreateServiceW
100. listen
101. NtQueryMultipleValueKey
102. DnsQuery_A
103. InternetWriteFile
104. InternetReadFile
105. FindFirstFileExA
106. NtCreateNamedPipeFile
107. FindFirstFileExW
108. closesocket
109. socket
110. RtlCreateUserThread
111. HttpSendRequestA
112. NtCreateSection
113. StartServiceW
114. SetWindowsHookExA
115. FindWindowExW
116. WSASendTo
117. InternetOpenA
118. VirtualProtectEx
119. SetWindowsHookExW
120. StartServiceA
121. HttpSendRequestW
122. LookupPrivilegeValueW
123. InternetOpenW
124. ExitWindowsEx
125. NtQueryValueKey
126. RegDeleteKeyA
127. HttpOpenRequestA
128. accept
129. FindWindowW
130. ControlService
131. RegDeleteKeyW
132. FindWindowA
133. HttpOpenRequestW
134. NtFreeVirtualMemory
135. InternetConnectW
136. UnhookWindowsHookEx
137. RegEnumKeyW
138. InternetConnectA
139. NtSaveKey
140. RegDeleteValueA
141. CopyFileExW
142. NtOpenMutant
143. NtOpenFile
144. RegQueryInfoKeyW
145. NtQueryDirectoryFile
146. NtDeleteKey
147. gethostbyname
148. CopyFileA
149. NtCreateMutant
150. GetAddrInfoW
151. InternetOpenUrlA
152. bind
153. NtOpenKey
154. OpenSCManagerW
155. InternetOpenUrlW
156. NtOpenKeyEx
157. OpenSCManagerA

APPENDIX D

FEATURES USED IN TRAINING

Features used for detection in Representation 1, with 1,000 malware samples, can be found in Table D.1.

1. API39	**18. API76**	**35. API10**
2. API69	**19. profapidll**	**36. API42**
3. SeqArg1	**20. API64**	**37. API12**
4. API134	**21. API66**	**38. API117**
5. API148	**22. API126**	**39. API58**
6. API157	**23. API14**	**40. API8**
7. API31	**24. API85**	**41. API78**
8. API41	**25. API56**	**42. API87**
9. API140	**26. API96**	**43. API130**
10. API5	**27. API55**	**44. API16**
11. Binservices	**28. shlwapidll**	**45. API59**
12. wininetdll	**29. API11**	**46. setupapidll**
13. API65	**30. API73**	**47. Binwindows**
14. API109	**31. API72**	**48. API32**
15. API94	**32. urlmondll**	**49. API118**
16. Binnetwork	**33. API34**	**50. API83**
17. API4	**34. API60**	

Table D.1: Features for Representation 1 with 1,000 malware samples.

Features used for detection in Representation 2, with 1,000 malware samples, can be found in Table D.2.

1. API39	**18. Binnetwork**	**35. API60**
2. API134	**19. API4**	**36. API8**
3. API69	**20. API64**	**37. API10**
4. Seq1	**21. Binprocess**	**38. API42**
5. API41	**22. shlwapidll**	**39. Binwindows**
6. API148	**23. profapidll**	**40. API117**
7. API157	**24. API76**	**41. API58**
8. API31	**25. API126**	**42. API112**
9. API109	**26. API78**	**43. API147**
10. API140	**27. API12**	**44. Binfilesystem**
11. API5	**28. API85**	**45. API144**
12. API65	**29. API56**	**46. API130**
13. wininetdll	**30. API96**	**47. API87**
14. Binservices	**31. urlmondll**	**48. API100**
15. API94	**32. API11**	**49. API59**
16. API14	**33. API72**	**50. setupapidll**
17. API66	**34. API34**	

Table D.2: Features for Representation 2 with 1,000 malware samples.

APPENDIX E

API LIST FOR MALWARE

API Name	Occurences	API Name	Occurences	API Name	Occurences
NtOpenKeyEx	123,803,970	**DeleteFileA**	1,349,497	**NtDeleteKey**	52,316
NtQueryValueKey	112,343,218	**CreateThread**	1,249,805	**NtSuspendThread**	50,233
LdrGetProcedureAddress	101,506,006	**RegDeleteValueW**	1,231,172	**accept**	47,631
GetSystemMetrics	86,133,172	**NtEnumerateValueKey**	1,025,003	**InternetOpenUrlA**	47,110
NtQueryKey	56,187,107	**select**	943,001	**GetAddrInfoW**	46,803
RegCloseKey	53,520,491	**WriteConsoleA**	665,686	**InternetReadFile**	43,933
RegOpenKeyExW	49,033,928	**ReadProcessMemory**	658,348	**NtSetContextThread**	38,304
RegQueryValueExA	30,247,859	**ExitThread**	635,374	**recvfrom**	37,216
NtQueryInformationFile	25,025,170	**NtSetValueKey**	610,604	**CreateRemoteThread**	36,980
RegQueryValueExW	24,425,249	**WriteProcessMemory**	580,392	**ShellExecuteExW**	35,902
NtDelayExecution	23,578,113	**NtResumeThread**	563,264	**NtCreateNamedPipeFile**	33,554
NtSetInformationFile	21,445,399	**recv**	506,197	**WSASocketW**	30,961
NtReadFile	21,293,746	**socket**	521,827	**InternetConnectA**	24,965
NtCreateFile	21,146,694	**NtOpenDirectoryObject**	497,575	**StartServiceA**	24,400
LdrLoadDll	18,424,469	**connect**	482,762	**HttpOpenRequestA**	23,487
RegOpenKeyExA	18,178,123	**DeleteFileW**	480,097	**HttpSendRequestA**	23,239
NtReadVirtualMemory	16,452,136	**closesocket**	463,183	**NtWriteVirtualMemory**	22,770
NtEnumerateKey	15,479,888	**anomaly**	440,898	**InternetOpenUrlW**	20,619
NtOpenFile	14,993,385	**VirtualProtectEx**	434,606	**CreateServiceA**	18,901
ZwMapViewOfSection	13,923,072	**RegDeleteValueA**	432,631	**InternetConnectW**	18,723
NtDeviceIoControlFile	13,671,481	**ExitProcess**	430,560	**HttpOpenRequestW**	18,651
NtOpenKey	13,377,187	**setsockopt**	380,427	**HttpSendRequestW**	18,584
FindWindowA	13,324,162	**FindWindowExW**	367,097	**sendto**	17,740
FindFirstFileExW	12,355,900	**ioctlsocket**	315,134	**MoveFileWithProgressW**	14,330
NtOpenMutant	11,131,074	**CopyFileA**	314,629	**CopyFileW**	12,812
LdrGetDllHandle	10,949,922	**gethostbyname**	311,989	**listen**	12,051
NtWriteFile	10,479,331	**OpenSCManagerA**	278,519	**WSASend**	11,050
RegEnumValueW	9,494,396	**WSAStartup**	275,992	**RegDeleteKeyW**	9,898
NtQueryDirectoryFile	8,905,546	**OpenServiceA**	268,999	**RemoveDirectoryW**	5,440
NtCreateSection	6,677,562	**send**	266,092	**NtSaveKey**	4,890
GetCursorPos	6,332,962	**OpenSCManagerW**	262,838	**NtTerminateProcess**	4,633
NtFreeVirtualMemory	5,863,378	**RegDeleteKeyA**	257,889	**NtTerminateThread**	4,284
NtProtectVirtualMemory	4,699,861	**OpenServiceW**	253,816	**StartServiceW**	2,452
FindWindowExA	4,448,321	**NtOpenThread**	211,568	**CopyFileExW**	1,977
RegSetValueExA	4,004,339	**RegQueryInfoKeyA**	210,412	**WSASocketA**	1,966
RegCreateKeyExW	3,511,255	**SetWindowsHookExW**	207,362	**system**	1,638
RegEnumKeyW	3,334,958	**getaddrinfo**	198,370	**DeleteService**	1,495
NtOpenSection	3,064,300	**NtCreateThreadEx**	161,894	**DnsQuery_A**	1,129
CreateDirectoryW	2,899,609	**InternetCloseHandle**	155,073	**ExitWindowsEx**	783
RegCreateKeyExA	2,786,542	**bind**	147,790	**WSARecvFrom**	434
CreateProcessInternalW	2,517,711	**RemoveDirectoryA**	130,098	**CreateDirectoryExW**	395
RegEnumValueA	2,491,840	**UnhookWindowsHookEx**	110,979	**NtDeleteValueKey**	338
RegEnumKeyExA	2,323,743	**LookupPrivilegeValueW**	107,869	**CreateServiceW**	306
NtCreateKey	2,200,170	**SetWindowsHookExA**	96,978	**FindFirstFileExA**	282
FindWindowW	1,912,765	**URLDownloadToFileW**	89,130	**RtlCreateUserThread**	205
RegSetValueExW	1,858,380	**WSARecv**	84,858	**InternetWriteFile**	148
WriteConsoleW	1,817,077	**shutdown**	73,446	**NtLoadKey**	77
RegEnumKeyExW	1,676,315	**NtGetContextThread**	72,976	**WSASendTo**	59
DeviceIoControl	1,634,702	**InternetOpenA**	70,145	**NtMakeTemporaryObject**	57
RegQueryInfoKeyW	1,479,221	**ControlService**	63,210	**NtQueryMultipleValueKey**	43
NtCreateMutant	1,479,090	**InternetOpenW**	53,564	**NtSaveKeyEx**	16

APPENDIX F

API LIST FOR CLEANWARE

API Name	Occurences
NtQueryValueKey	2,062,544
NtOpenKeyEx	1,437,612
NtQueryKey	681,181
RegEnumKeyW	192,503
LdrGetProcedureAddress	185,995
RegOpenKeyExW	117,556
NtCreateFile	71,382
RegCloseKey	70,534
GetSystemMetrics	52,363
NtOpenKey	50,363
NtQueryInformationFile	46,078
LdrGetDllHandle	41,910
RegEnumValueW	40,780
LdrLoadDll	39,567
NtQueryDirectoryFile	39,401
NtReadVirtualMemory	33,049
RegQueryValueExW	30,970
NtReadFile	30,277
FindFirstFileExW	29,565
NtDeviceIoControlFile	28,624
ZwMapViewOfSection	24,872
NtProtectVirtualMemory	23,343
NtEnumerateKey	17,115
NtOpenFile	15,918
RegQueryInfoKeyW	14,548
NtCreateSection	14,270
NtOpenSection	13,469
RegSetValueExW	10,988
NtSetInformationFile	8,130
RegQueryValueExA	7,131
RegDeleteValueW	6,056
NtFreeVirtualMemory	5,712
ReadProcessMemory	5,029
NtCreateMutant	4,762
NtWriteFile	4,577
anomaly	4,067
DeviceIoControl	3,861
WriteConsoleA	2,401
GetCursorPos	1,796
NtDelayExecution	1,777
NtResumeThread	1,736
RegEnumKeyExW	1,714
OpenServiceW	1,694
NtOpenDirectoryObject	1,621
CreateProcessInternalW	1,590
ExitThread	1,587
NtCreateKey	1,389
OpenSCManagerW	1,379
StartServiceW	1,336
NtOpenMutant	1,258
RegOpenKeyExA	1,254

API Name	Occurences
RegCreateKeyExW	1,227
listen	1,211
accept	1,210
CreateThread	1,206
select	1,163
CreateDirectoryW	1,156
NtOpenThread	1,044
NtCreateThreadEx	837
WriteConsoleW	665
FindWindowA	456
RegCreateKeyExA	453
bind	70
NtEnumerateValueKey	414
RegEnumKeyExA	290
RegEnumValueA	271
SetWindowsHookExA	269
setsockopt	232
NtSetValueKey	204
UnhookWindowsHookEx	201
ExitProcess	143
WSAStartup	126
NtSuspendThread	125
RegSetValueExA	114
FindWindowW	109
SetWindowsHookExW	103
CopyFileA	98
DeleteFileW	88
InternetCloseHandle	84
closesocket	69
getaddrinfo	67
recv	62
send	62
WSARecv	56
socket	43
HttpOpenRequestW	40
HttpSendRequestA	39
InternetConnectW	38
connect	36
ioctlsocket	36
NtGetContextThread	36
WSASend	34
GetAddrInfoW	34
WSASocketW	32
DeleteFileA	28
LookupPrivilegeValueW	19
OpenSCManagerA	17
FindWindowExA	16
FindWindowExW	16
OpenServiceA	15
MoveFileWithProgressW	14
NtCreateNamedPipeFile	14

API Name	Occurences
shutdown	12
RegDeleteValueA	8
RegQueryInfoKeyA	7
NtDeleteKey	6
WriteProcessMemory	6
NtTerminateThread	4
CopyFileW	4
gethostbyname	4
RemoveDirectoryW	3
ShellExecuteExW	3
StartServiceA	3
CreateServiceA	3
RegDeleteKeyA	2
DeleteService	2
WSASocketA	2
RegDeleteKeyW	2
HttpSendRequestW	1
RemoveDirectoryA	1
InternetReadFile	1
InternetOpenW	1

INTENTIONALLY LEFT BLANK

Analysis of Malware Behavior: Type Classification using Machine Learning

Radu S. Pirscoveanu, Steven S. Hansen
Thor M. T. Larsen, Matija Stevanovic, Jens Myrup Pedersen
Aalborg University, Denmark
Email: rpirsc13@student.aau.dk, ssha10@student.aau.dk,
tmtl10@student.aau.dk, mst@es.aau.dk, jens@es.aau.dk

Alexandre Czech
Ecole Centrale d'Electronique
Paris, France
Email: aczech@ece.fr

***Abstract*—Malicious software has become a major threat to modern society, not only due to the increased complexity of the malware itself but also due to the exponential increase of new malware each day. This study tackles the problem of analyzing and classifying a high amount of malware in a scalable and automatized manner. We have developed a distributed malware testing environment by extending Cuckoo Sandbox, that was used to test an extensive number of malware samples and trace their behavioral data. The extracted data was used for the development of a novel type classification approach based on supervised machine learning. The proposed classification approach employs a novel combination of features that achieves a high classification rate with weighted average AUC value of 0.98 using Random Forests classifier. The approach has been extensively tested on a total of 42,000 malware samples. Based on the above results it is believed that the developed system can be used to pre-filter novel from known malware in a future malware analysis system.**

Keywords: Malware, type-classification, dynamic analysis, scalability, Cuckoo sandbox, Random Forests, API call, feature selection, supervised machine learning.

February 25, 2015

I. Introduction

The trend of the Internet usage has grown exponentially in the past years as modern society is becoming more and more dependent on global communication. At the same time, the Internet is increasingly used by criminals and, a large black market has emerged where hackers or others with criminal intent can purchase malware or use malicious services for a renting fee. This provides a strong incentive for the the hackers to modify and increase the complexity of the malicious code in order to improve the obfuscation and decrease the chances of being detected by anti-virus programs. This leads to multiple forks or new implementations of the the same type of malicious software, that can propagate out of control. Based on AV-Test, approximately 390,000 new malware samples are registered every day, which gives rise to the problem of processing the huge amount of unstructured data obtained from malware analysis [2]. This makes it challenging for anti-virus vendors to detect zero-day attacks and release updates in a reasonable time-frame to prevent infection and propagation.

Meeting this problem, researchers and anti-virus vendors seek towards finding a faster alternative method of detection that can overcome the limitations imposed by static analysis, which is the classical approach. Analyzing the malicious code can yield inaccurate information when polymorphic, metamorphic and obfuscating methods are used. When aforementioned methods are applied the complexity increases even more, thus it will be hard to determine which type of malware it is. An alternative to the approach presented, is performing dynamic analysis on the behavior of the malicious software which can also be a troublesome task when having to analyze an extensive and increasing number of new malware. Due to these problems it is therefore favorable to develop a scalable setup where several malware can be dynamically analyzed in parallel. A large amount of malware samples have been utilized compared to past researched articles for this study. Having a large sample-set adds up to the predictive power and reliability of the built classifier which provides satisfactory results. In this study, a system has been developed which could be used as a pre-filtering application, where all known types can be sorted from the novel malware. This leaves the opportunity to skip static analysis on known malware and focus only on analyzing the novel malware, thus drastically increasing the detection and analysis rate of anti-virus programs. New malware that arise each day are believed to be mostly modified versions of previous malware, using sophisticated reproduction techniques. Stating this, it is assumed in this study, that malware, even though it is new, can exhibit similar behavior as earlier versions from a dynamic analysis point of view. [12]

This study is based on a university report written by this group in [3]. In section II the background and discussion about improvements of related work are presented, followed by the methodology in section III proposing a solution for the problems presented in the introduction. Finally the results and conclusion will be presented in section IV and section V respectively.

II. Related Work

When classifying malware types it is essential to find parameters that can distinguish between their behavior, where commonly used parameters on Windows platforms are the Windows API calls. The reason that these are commonly used is that they include a solid and understandable form of behavioral information since an API call states an exact action performed on the computer, e.g. creation, access, modifica-

tion and deletion of files or registry keys. In [10] they use hooking of the system services and creation or modification of files. Additionally they use logs from various API calls to differentiate malware from cleanware as well as performing malware family classification. They include a sample set of 1,368 malware and 456 cleanware where they use a frequency representation of the features. The limitation, also emphasized in their future work, is that they need to expand their sample set and explore new features. In [16] they made a scalable approach using the API names and their input arguments, after which they applied feature selection techniques to reduce the number of features for a binary classifier that includes the separation of malware and cleanware. The features used in their setup are limited to features related to the API system calls during run-time. Here they have a sample set of 826 malware and 385 cleanware. Additionally they apply a frequency representation, as the research mentioned before in [10], but also include a binary representation. Furthermore in [5], they use CWsandbox, which applies a technique called *APIhooking* to catch the behavior of the malware, but in this paper they strive to classify malware into known families. They here use a total sample set of 10,072 malware and utilize a frequency representation of their features. In terms of automatic analysis, [6] has created a framework able to perform thousands of tests on malware binaries each day. Here they use a sample set of 3,133 malware and use a sequence representation of their features, which here are the Windows API calls applied for both clustering and classification. To understand how API calls are used by malicious programs, [8] have made a grouping of features in relation to their purpose, which can be helpful to understand the malware behavior. In terms of classification approaches, a wide range of machine learning algorithms are used such as J48, Random Forests and Support Vector Machine. The weakness of the related work is the limited amount of samples used to build their classifier. Furthermore this study propose a feature representation that combines several of the aforementioned representations to achieve a greater behavioral picture of the malware.

Given that the labels for malware types are provided by anti-virus vendors and based on the related work, it is found that supervised machine learning is a valid choice for this study. Based on a dataset generated from around 80,000 malware samples, a feature selection has been performed after analyzing the data. In the mentioned research articles, API calls are the mainly used parameter for creating features. In this study several parameters were chosen as features in addition to API calls. The additional parameters are: mutexes, registry keys/files accessed and DNS-requests. In the related work, different feature representations were used, i.e. sequence, binary and frequency. The contribution of this study is the unique combination of different feature representations and parameters that also apply feature reduction strategies. Furthermore our study includes a great amount of malware samples and behavioral data collected using our setup. This allows a solid basis when training the model since it includes a larger behavioral picture of the malware. Finally, we rely on Random Forests classifier to perform the classification of the malware types, as a capable ensemble classifier also used by related work [10], [16].

III. Methodology

This section will go through the methodology applied in the development of this study. This includes: Dynamic analysis, supervised machine learning, data generation, data extraction and classification.

A. Dynamic Analysis

As mentioned earlier, large amount of malware are injected into the Internet every day, which makes it more and more suitable to use a dynamic approach in contrast to static analysis. Dynamic analysis is performed in such a way that malware is executed in a sandbox environment in which it is assumed that malware believes it is on a normal machine. Here, all actions performed at run-time, are recorded and saved in a database. This is different from the classical signature-based approach also used in the context of static analysis that is commonly applied by anti-virus vendors. In this study, Cuckoo Sandbox has been chosen as the sandbox environment in which the malware will be injected, see [4]. Since Cuckoo is open source, it allows to openly modify the software, which means it is possible to change the code to fit the needs of this study. One of the requirements is to make the system distributed and scalable, such that it can be controlled from one central unit and new virtual machines or physical machines can easily be added in order to improve the efficiency of the overall analysis.

B. Supervised Machine Learning

Using dynamic analysis to gather behavioral data, it is possible to perform malware type classification using supervised machine learning. We have chosen Random Forests with 160 trees, which is a decision tree based algorithm that makes use of random sub-sampling, or tree bagging, of the sample space that are then used to create a tree for each subset [7]. Individual decision making is utilized at each tree for each classification of malware, where the results are then averaged. This prevents the possibility of over-fitting, as variance of the classification model decreases when averaged over a suitable amount of trees. In this study the machine learning tool WEKA has been used, which can be run through java-based GUI or directly in the terminal [15].

In Figure 1 an overview of the system is depicted as a flowchart. It includes modules for each of the groups: Data Generation, Data Extraction and Malware Classification. Each group will be explained in the following subsections.

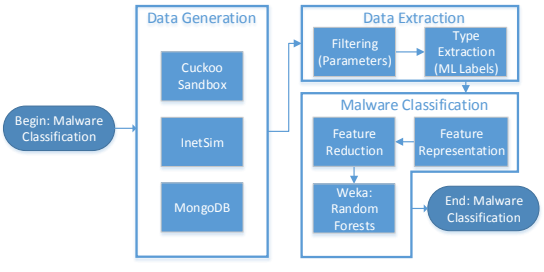


Fig. 1. Overall system flow.

C. Data Generation

The data generation consists of generating malware analysis reports from the execution of approximately 80,000 malware samples downloaded from Virus Share [13]. To perform the analysis in a secure, scalable and distributed environment, a customized system has been set up.

The designed system consists of a modified version of Cuckoo Sandbox [4] which permits to perform a faster analysis based on parallel computing. It is a distributed virtual environment composed of 13 personalized guests and a control unit. In order to simulate a real environment, the malware is executed within a personalized installation of Microsoft Windows 7 operating system. Some commonly used software are installed (Skype, Flash, Adobe Reader, etc.) along with a batch script that simulates web activity. The modifications made permit to obtain a distributed system which is also scalable, making it possible to easily add or remove virtual machines. Moreover, it has been noticed that during their execution, some malware intended to connect to the Internet. This raised security requirements related to potential malicious traffic activity. The challenge has been to emulate a realistic environment without allowing any malware to communicate with a third party. To complete the security needs, a confined environment has been built. We configured InetSim, an Internet emulator, so that it responds to the malware requests and deceive the operating system into perceiving that it is online [11], [9]. On the other hand, the internal system configuration permits to avoid the corruption of the analysis environment. This is why the virtual machines and their hosts are running on two different operating systems. To enhance security, all the commands are sent from the control unit to the hosts using Secure Shell.

Finally, the collected data, consisting of recorded malware actions, is stored using the DataBase Management System, i.e. MongoDB. This type of DBMS is particularly useful in dealing with a large database that includes unstructured data which is the case in this study.

D. Data Extraction

The data extraction consists in keeping the most pertinent information to be used in a machine learning algorithm. First, the reports provided by Cuckoo give a wide set of information about the malware behavior, namely: DNS requests, Accessed Files, Mutexes, Registry Keys and Windows API's. To be sure that this information is only related to malicious activity, a white list is created to clean the data from the non malicious activity. It consists in recording the user's activity simulated by a batch script that executes programs and browse the web. Afterwards this behavioral data is removed from the malware analysis reports.

Parameters	Before filtering	After filtering	Percentage Decrease
DNS (all levels)	2,000	1,986	0.7 %
DNS (TLD & 2-LD)	1,549	1,543	0.39 %
Accessed Files	673,554	18,322	97.28 %
Mutexes	11,287	9,875	12.51 %
Registry Keys	164,979	94,505	42.71 %

TABLE I. SUMMARY OF FILTERING RESULTS.

Table I lists the different parameters that have been filtered from the analysis. One can see that no filtering is applied on the API calls since they are logged based on the process ID of the malware.

Dealing with the great amount of unstructured data from the performed dynamic analysis, it gives rise to the problem of selecting features that precisely distinguish the types. In this study, a big effort has been put into primarily analyzing the behavioral data given by the API calls as in [10], [16], [5], [6] and [8]. Each API call corresponds to a specific action performed on the system that permits to characterize the malware behavior which is the reason why it has been decided to choose the API as the main parameter. Nevertheless, we have chosen to use the other parameters to play a complementary role in the malware classification.

The second step consists in labeling the samples to be used in supervised machine learning. Once the analysis is done, Cuckoo Sandbox provides a report that includes a list of anti-virus programs along with the corresponding labels. The challenge is to find the anti-virus program that gives both the best detection rate and the most precise labeling. Given these requirements, VirusTotal, [14], provided labels for around 52,000 samples based on detection from Avast [1]. From the sample set, four different types were detected, represented by 42,000 suitable samples for classification, namely:

Trojan: includes another hidden program which performs malicious activity in the background.

Potentially Unwanted Program: is usually downloaded together with a freeware program without the user's consent, e.g. toolbars, search engines and games.

Adware: aims at displaying commercials based on the user's information.

Rootkit: has the capability to obfuscate information like running processes or network connections on an infected system.

E. Malware Classification

This section will go through the Feature Representation and Feature Reduction that are the two preliminary steps before using WEKA toolbox to both perform classification and measure the predictive performances of the training model.

1) Feature Representation: The built features tend to give a meaning to the chosen parameters. The total number of 151 different API calls are the main features, whereas complementary information is derived from the other parameters. The features are gathered within a matrix where each row represents a malware sample and each column gives the corresponding value of a specific feature. These are the different representations:

Sequence: For each sample the course of the 200 first API calls during the malware execution, is used. This number has been chosen to obtain a reasonable matrix size. Besides, the initial sequence of API calls has been modified to improve the matching between malware that have similar patterns. The interest of this modification is illustrated in Figure 2.

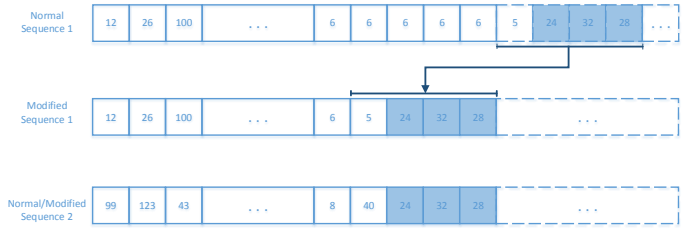


Fig. 2. Sequence Modification.

Since the initial sequence size has been limited to 200 API calls, it is likely that the repetition of the same API hides patterns that are out of the scope. Thus, to retrieve eventual hidden similarities, the sequence is modified so that it gives the succession of actions performed without taking care of their frequency. It is done by removing the repetition of the same API called in a row.

Frequency: This matrix is composed of 151 columns corresponding to the set of API calls. The frequency of each API call is calculated from the malware analysis.

Counters: This matrix is composed of 8 columns which corresponds to the count of the 8 following parameters: DNS request (all levels), DNS request (TLD and 2-LD), Accessed Files (including 3 file extensions), Mutexes and Registry Keys.

2) Feature Reduction: In order to perform efficient large scale analysis by combining different behavioral features, two dimensionality reduction methods are used. The first one is applied on the sequence matrix and consists in reducing the initial sequence length by observing the impact on the classification's performance. Here, we have kept 40 features since we found that it contains substantial information to classify malware types. In addition, it has been chosen to combine the frequency of the 151 API calls into bins of the same category inspired from [8]. Thus, 24 bins are created and can be grouped into 7 categories (Registry Management, Windows Services, Processes etc).

The challenge of the feature reduction is to minimize the number of features without loosing the performance of the classification. The individual reduction of features aims at limiting the final number of features within the combination matrix.

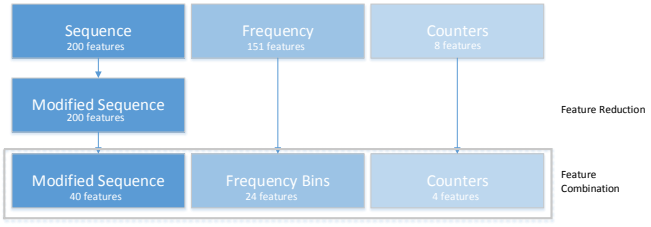


Fig. 3. Construction of the combination matrix.

Figure 3 shows the transformations performed to build a matrix which is a combination of the different features. The model conceptually gives a more and more general information about the malware behavior. It is noticeable that the sequence is modified and reduced so that the new sequence gives the course of the 40 first actions without repetitions performed during the malware analysis. Afterwards, the frequency of the bins gives the occurrence of the 151 API calls grouped by type, during run-time. Finally, the counters provide the most general information since they give the occurrence of the complementary parameters but without indication of the action performed.

IV. Results and Discussion

The classifier is configured through a development and training phase, after which it is tested, producing results that will be evaluated in this study. The total number of samples being used for the training and testing phase are 42,068 samples, from which 67 % represents the training set and the remaining 33 % represents the testing set. In the following, the development and training phases will be presented, along with the results of the classification.

A. Development and Training

The development phase is used to decide the number of trees that should be used in the Random Forests algorithm. Here, 160 trees were chosen to provide a good balance between improved results and computational time. The results for the development phase were evaluated using the training set with 10-fold cross-validation. After deciding the parameters for the RF algorithm, a training phase was used to construct the model used to classify the four different types of malware presented in section III. The training phase also had a second purpose, namely to choose the feature representation that should be used to configure the classifier, since multiple representations have been examined in this study. The Area Under the Curve (AUC value) and F-measure are provided by a 10-fold cross-validation for different feature representations. These are trained with a different number of features whereas an objective choice was made based on the results to construct a matrix by combining multiple feature representations. Based on the results, the combined feature representation was chosen, as it gave the best AUC value and F-measure compared to the other representations examined. The combination will include the 40 first distinct API calls, 24 frequency bins and 4 counters namely the count of distinct mutexes, files, registry keys and all levels of the DNS.

B. Results

The results from the testing phase will be presented in the form of a table with the most important available metrics, together with ROC curves and a confusion matrix. In Table II the True Positive Rate (TPR), False Positive Rate (FPR), Precision, F-measure and AUC value can be found for each class/type. To summarize the results from the table, ROC curves can be found in Figure 4 and a confusion matrix in Figure 5. Below, each class will be analyzed based on the results found in Table II, Figure 4 and Figure 5.

1) Trojan: Based on the results, the classifier revealed the best performance for this type of malware, which also can be due to the fact that it has the largest amount of samples compared to the other three types examined. With the classifiers high precision and F-measure of respectively 0.961 and 0.960, it shows promising classification results for this

type. This conclusion is also supported by the high AUC value of 0.989. Looking at the ROC curve in Figure 4, it can be seen to be well behaving and steep, leading to a high discriminative power. Looking at the confusion matrix in Figure 5 it can be seen that the classifier has a potential problem to distinguish Trojan from Adware. It should be noticed that the number of FNs is small compared to the number of Trojan samples.

2) Potential Unwanted Programs - PUP: The classifier shows a good classification performance for PUP in comparison with Trojan. Looking at Table II, the precision and F-measure are 0.939 and 0.850 respectively. These values are lower than the results for Trojan, but overall the performance of the classifier is still satisfactory. The lower F-measure is caused be the TPR, which is lower than the precision. Looking at the ROC curve it is seen to be well behaving, but not as steep as Trojan. This is expected from the results of the table, however PUP has an AUC of 0.978, which is considered satisfactory. Presenting the confusion matrix in Figure 5, it can be noticed that the classifier mostly confuses PUP with Adware, having 672 FNs.

3) Adware: Before the test, some behavioral similarities were expected between Adware, Trojan and PUP, since these malicious programs infiltrate the infected machine using common methods, however with a different end goal. The precision and F-Measure of Adware, seen in Table II have the lowest value of all tested types due to the large number of FPs which also results in an FPR of 0.085 and a number FNs that push the TPR to 0.858. The AUC value is 0.955 which is also the lowest of all four types. By looking at the ROC curve in Figure 4, even though it is far beyond the theoretical ROC curve of a random classifier (blue dashed line), more information might be needed to be able to better discriminate this type. The confusion matrix in Figure 5 generated by the classifier, shows that a large portion of the Adware samples have been correctly classified but with approximately one forth of the samples being classified as PUP or Trojan. PUPs can be classified as Adware depending on the severity of the logging or content presented to the user, however due to the large sample size of Trojan in comparison with other types, the TNs remain very high thus the FPs' number is shadowed by the TNs resulting in a low FPR.

4) Rootkit: It represents one of the most distinct malware types in comparison with Trojan, Adware and PUP. The action it performs on the infected PC should present a behavioral pattern that can easily be distinguished as it tries to mask itself inside system components. Even though the number of samples in training and testing is significantly lower than other types, its unique behavior resulted in a precision of 0.947 which represents the second highest value of all tested types. However the F-measure has a value of 0.862, which is due to the lower TPR. The FPR has a value close to 0 due to the large number of TNs and a low value of 8 for the FPs. The FNs of Rootkit represent a large number in comparison to the low number of samples resulting in a lower TPR, which affected the F-measure. The ROC curve presents a high discriminative power in comparison with the other presented types with an AUC value of 0.970, which is closer to the values of the types that have a more dominant number of samples.

5) Summary of results: The weighted average calculated of all types reveals a high discriminative power of the classifier in terms of the AUC value. As discussed before, the FPR becomes low since the TNs are much larger in contrast to the FPs. The weighted average of the F-measure shows less discriminative power as PUP and Rootkit types have a high number of FNs in comparison with the number of samples. Overall the classifier has a satisfactory performance with an AUC value close to the theoretical maximum and an F-measure just below 0.9.

Class	TPR	FPR	Precision	F-measure	AUC
Trojan	0.959	0.052	0.961	0.960	0.989
PUP	0.777	0.015	0.939	0.850	0.978
Adware	0.858	0.085	0.693	0.767	0.955
Rootkit	0.791	0.001	0.947	0.862	0.970
Weighted Avg.	0.896	0.049	0.907	0.898	0.980

TABLE II. RESULTS OF TESTING PHASE.

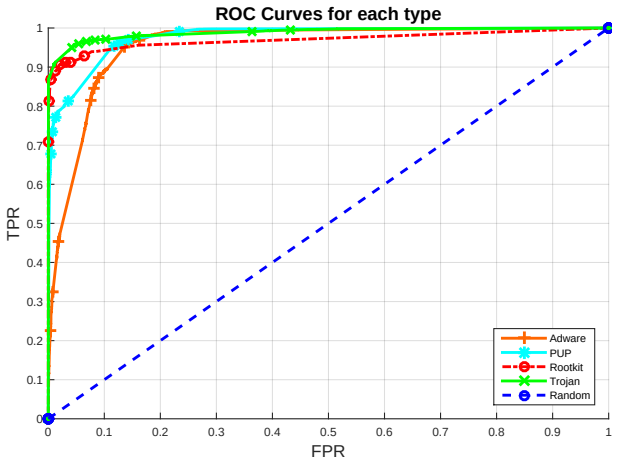


Fig. 4. Receiver Operating Characteristics for the classified types.

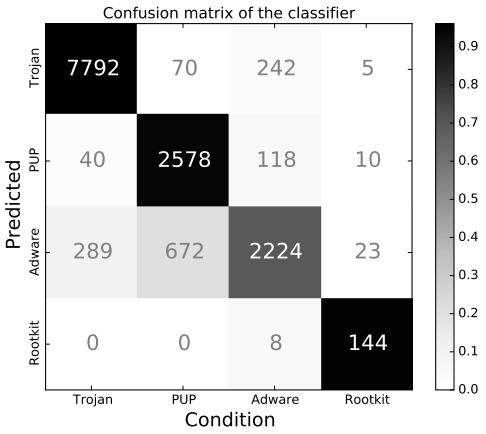


Fig. 5. Results of the classifier in the form of a confusion matrix.

C. Discussion

This section elaborates on the obtained results assessing if the results for the particular malware type are good enough to be used in a future system that can pre-filter newly registered

malware samples. It should be noticed that future work will be devoted to optimize the classifier such that a pre-filtering system can be developed to identify novel malware samples and sort out legacy malware that have minor changes.

Trojan - The problem of having a small number of Trojan samples classified as Adware can be caused by the fact that some of the Adware samples actually are Trojans, which have been used to install the Adware while running the experiment. With this small amount of FPs the classifier still performs satisfactory for Trojan and it can in fact be used as pre-filtering for this type.

PUP - Has a definition that may include other types. It could be classified as Adware depending on the amount of content presented to the user. From the analysis of the results a certain amount was classified as Adware. This problem can be caused by the fact that Adware and PUPs are similar, since PUPs are just a less severe case of Adware, making a common behavior possible. The classifier performs well for this type of malware and it is possible to use it as pre-filtering.

Adware - Similarities between the behavior of Adware and PUP shown by the classifier might be an incentive to retrieve even more detailed information about API calls. The classifier for this type performs fairly good, but needs to be improved before it can be used as pre-filtering with satisfactory results. It was found that the label of Adware from Avast was unclear and too generic, which could explain the results. This problem can be solved by a more clear and categorized definition of this type.

Rootkit - The classifier performs well regardless of the low amount of samples, which could be because of its distinct behavior. Therefore this classifier can be used as pre-filtering, but in order to ensure a good performance, more samples should be used to train the model.

Using the system as a pre-filter - Based on the individual and weighted average results, it can be concluded that the system created can be used in part of a pre-filtering application. It will work by rejecting malware as novel, when the malware can not be classified as any type with a high enough probability. The results for the individual types are satisfactory for every type except Adware, which is the only type that pulls down the performance if looking at the weighted average results.

D. Future Research

In order to improve the presented classification of malware types, future research must done to a achieve an even better discrimination. Having a uniform distribution of all malware types can improve the results of the classifier by assigning the same weight on all samples instead of favoring Trojan due the clear advantage in samples size. This will affect the classifier by having a fair distribution in the TNs for each type and assigning the statistics for TPR and TNR over an uniform number of samples. In this particular case, the number of FNs, FPs and TPs should not shadowed by a large number of TNs. That being said, a more detailed approach in building the API features can be taken by looking at the arguments passed during the API calls. This could give a more detailed view of the malicious behavior expressed by the samples. The accuracy of the pre-filtering system can be improved by theoretically having behavior information for all existing types, thus allowing the detection of novel malicious software by using a probability threshold.

V. Conclusion

This study proposes a novel malware classification approach developed in order to provide an accurate classification of malware types within the dynamic analysis of malware. It relies on a novel set of features that successfully capture differences in the behavior of malware types. Starting from the estimation made by AV-Test where approximately 390,000 new malware are released every day, the proposed malware analysis approach aims to provide a pre-filtering solution to this problem in order to distinguish novel malicious software, that has significantly different behavior from malware known by the classifier. Having three main modules: Data Generation, Data Extraction and Malware Classification, this study provides a fast, distributed and secure method of analyzing malware with high predictive performance.

The combination of features from approximately 80,000 samples consists of: 24 API Frequency Bins, 40 Modified Sequence and four Counters collected using a modified version of Cuckoo Sandbox. The combination of behavioral information proved to be very detailed allowing us to detect the correct types after passing it through Random Forests algorithm with 160 trees. The weighted average results gathered using the novel feature representation, are very satisfactory. Having a steep Receiver Operator Curve, an Area Under the Curve of 0.98 (close to the theoretical maximum), a precision of 0.9 and an F-measure of 0.898, the classifier has proven to have a high discriminative and predictive power, which can be used to filter novel from know malware.

Acknowledgments

During this research project we received substantial help from VirusTotal by providing access to their vast database containing labels from 56 anti-virus programs that allowed us to perform classification as accurately as possible. Virus Share by providing access to their library of novel and known malicious programs from which we have collected the behavioral data.

References

[1] Avast. "Avast 2015." Internet: https://www.avast.com, 2015 [Feb. 22, 2015].

[2] AV-test, *Malware*, 2014. The Independent IT-Security Institute 2014. http://www.av-test.org/en/statistics/malware/ [Feb. 22, 2015].

[3] A. Czech, R. S. Pirscoveanu, S. S. Hansen and T. M. T. Larsen *Analysis of Malware Behavior: Type Classification using Machine Learning*, 2014 Aalborg, Denmark.

[4] Cuckoo Foundation. "Automated Malware Analysis - Cuckoo Sandbox." Internet: http://www.cuckoosandbox.org/, 2014 [Feb. 22, 2015].

[5] K. Rieck, T. Holz, C. Willems, P. Düssel and P. Laskov *Learning and Classification of Malware Behavior*, 5th Int. Conf. DIMVA 2008 Paris, France: Springer, 2012.

[6] K. Rieck, P. Trinius, C. Willems and T. Holz *Automatic of Analysis of Malware Behavior using Machine Learning*, 19th Vol. Issue 4 Jour. of Comp. Sec. 2011 Amsterdam, The Netherlands: IOS Press Amsterdam, 2012.

[7] L. Breiman. (2001, Jan.). *Random Forests*. [Online]. Available:http://oz.berkeley.edu/~breiman/randomforest2001.pdf [Feb. 22, 2015].

[8] M. Alazab, S. Venkataraman and P. Watters, *Towards Understanding Malware Behaviour by the Extraction of API calls*, 2nd CTC 2010 Ballarat (VIC), Australia: IEEE, 2012.

[9] M. Platts. "The Network Connection Status Icon." Internet: http://blogs.technet.com/b/networking/archive/2012/12/20/the-network-connection-status-icon.aspx, Dec. 20, 2012 [Feb. 22, 2015].

[10] R. Tian, R. Islam, L. Battern and S. Versteeg, *Differentiating Malware from Cleanware Using Behavioral Analysis*, 5th Int. Conf. on Malicious and unwanted Software Nancy, France: IEEE, 2010.

[11] T. Hungenberg, M. Eckert. "INetSim: Internet Services Simulation Suite." Internet: http://www.inetsim.org/, 2014 [Feb. 22, 2015].

[12] U. Bayer, E. Kirda, C. Kruegel *Improving the Efficiency of Dynamic Malware Analysis*, 25th Symposium On Applied Computing (SAC), Track on Information Security Research and Applications Lusanne, Switzerland, 2010.

[13] VirusShare. "VirusShare.com - Because Sharing is Caring." Internet: http://virusshare.com/, Feb. 22, 2015 [Feb. 22, 2015].

[14] VirusTotal. "virustotal." Internet: https://www.virustotal.com/, 2015 [Feb. 22, 2015].

[15] WEKA. "Weka 3: Data Mining Software in Java." Internet: http://www.cs.waikato.ac.nz/ml/weka/, 2014 [Feb. 22, 2015].

[16] Z. Selehi, M. Ghiasi and A. Sami, *A miner for malware detection based on api function calls and their arguments*, 16th AISP 2012 Shiraz, Fars: IEEE, 2012.