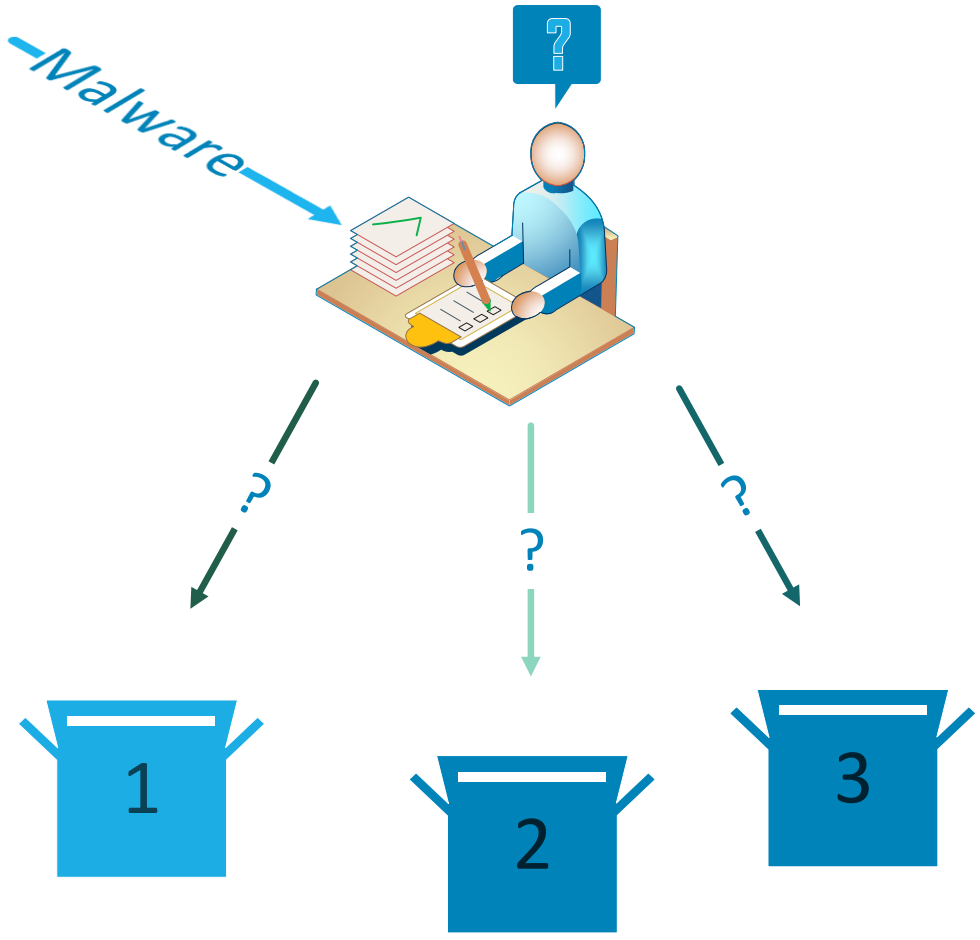
MASTER THESIS
Clustering Analysis of Malware Behavior



Aalborg University
P10, spring semester 2015
Group 1020 - 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:
Clustering Analysis of Malware Behavior

Theme:
Networks & Distributed Systems

Projectperiod:
P10, spring semester 2015

Group:
1020

Members:

Radu-Stefan Pirscoveanu

Supervisors:
Jens Myrup Pedersen
Matija Stevanovic

Number of pages: 139

Number of appendix pages: 16

Number of annex pages: 1 (CD)

Finished: 2nd of June 2014

Abstract:

At present, behavioral classification of malware is realized by means of Antivirus generated labels. This study investigates the inconsistencies associated with current practices by using unsupervised learning on malware behavior. Based on the problem isolation, research was undertaken to determine how Antivirus vendors label detected malware, as well as to raise the problem of inconsistency in their labeling results. A customized version of Cuckoo Sandbox was used to collect actions from approximately 270,000 malware samples, and to create their behavioral profile consisting of Passed and Failed API calls and their respective Return Codes.
Evaluating the detection results of Antivirus vendors on Completeness, Consistency and Correctness, and based on the devised analysis, a temporary solution was depicted, which involved performing a Majority Vote between multiple vendors. A tokenized Levenshtein ratio was used, in order to implement the vote and determine the appropriate labels for evaluation. Following close examination of the limited amount of options present in unsupervised Machine Learning for Feature Selection and optimal number of clusters, it was decided to make use of Principal Component Analysis along with Gap Statistics. The Self Organizing Map algorithm, preferred for clustering the behavioral data, provided an innovative approach for preserving the topological properties of the higher dimensionality information present in the malware dataset.
Upon evaluation of the Self Organizing Map clusterer, and taking into consideration the limited range of tools provided by unsupervised learning, the study showed shortcomings when relying on AV vendors for labeling malware samples. This is an indication, that no link exists between AV extracted type labels and generated behavioral clusters. To solve this discrepancy, a cluster-based classification is proposed, that is able to accurately classify new mailicious software using the clusters created with Self Organizing Map.

Preface

This Master Thesis has been written by group 1020, represented by a 4th semester student in the Networks & Distributed Systems masters program under the Department of Electronic Systems at Aalborg University. The subject of the Master Thesis is *"Clustering Analysis of Malware Behavior"*. The thesis 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 presumes the reader to have a basic knowledge within the field of networking, programming and malware analysis.

This report includes 7 Parts. It is built up by the following: Problem Analysis, Technical Analysis, Design & Implementation, Results, Conclusion, References and Appendices. 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 9 and will include information about the source, 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 are included after the table of contents.

INTENTIONALLY LEFT BLANK

Contents

INTENTIONALLY LEFT BLANK

Abbreviations

API: Application Programming Interface

AUC: Area Under the Curve

AV: Anti Virus

BMU: Best Matching Unit

BSON: Binary Script Object Notation

C&C: Command and Control

CSV: Comma Separated Values

DBMS: Database Management System

DBSCAN: Density-Based Spatial Clustering of Applications with Noise

DDoS: Distributed Denial of Service

DLL: Dynamic Link Library

DNS: Domain Name Service

DoS: Denial of Service

DRV: Driver

EM: Expectation-Maximization

FPR: False Positive Rate

GB: GigaBytes

GMM: Gaussian Mixture Model

HDD: Hard-Drive Disk

HTTP: HyperText Transfer Protocol

ID: Identification

IRC: Internet Relay Chat

JSON: JavaScript Object Notation

MB: MegaBytes

ML: Machine Learning

Mutex: Mutual Exclusive

NoSQL: Not Only Structured Query Language

OCX: Object Linking and Embedding Control Extension

OPTICS: Ordering points to identify the clustering structure

OS: Operating System

PC: Personal Computer

PCA: Principal Component Analysis

PID: Process ID

PUP: Potentially Unwanted Program

RAID: Redundant Array of Independent Disks

RAM: Random Access Memory

RF: Random Forests

ROC: Receiver Operator Characteristics

SOM: Self-Organizing Map

SP: Service Pack

TB: TeraBytes

UAC: User Account Control

UI: User Interface

VM: Virtual Machines

INTENTIONALLY LEFT BLANK

Part I

Problem Analysis

INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Malware infections have been one of the main concerns in the security community for the past years. With an increasing number of new malicious programs, Anti Virus (AV) vendors try to keep up with the trend in order to protect an also increasing number of computer users. As detection is becoming more and more complex due to the increased complexity of malicious code, a more advanced technique to detect and combat these threats needs to be researched. Following this direction, researchers have shifted their focus from traditional static based methods of detection [Sharif et al., 2008], [Moser et al., 2007] to more complex, dynamic and automatized solutions based on collecting malware behavior traces [Egele et al., 2012], [Ahmad et al., 2015], [Gorecki et al., 2011].

In our previous paper [Pirscoveanu et al., 2015], it has been attempted to solve the problem of classifying the increasing number of malicious software that emerge each day. Collecting information from approximately 80.000 malware samples, the information was carefully analyzed using novel feature selection and representation techniques. Using classification on the behavioral data, the study has shown that the implemented system can indeed classify known malicious types with high accuracy taking a step forward in the research of stopping infections using behavioral analysis. However, the method used, was dependent on information collected from AV vendors and the success of the study relied more on the correctness of the statically generated labels and assumed that there exists a link between static AV collected data and behavioral data collected using dynamic methods.

Attempting to understand the importance of correct labeling of detected malware provided by AV companies, a study made by [Mohaisen et al., 2014] has shown that there exist significant differences between methods used. This can result in labeling inconsistencies that may prevent AV vendors from correctly detecting new and updated malware or completely cleaning an infected machine. The labels represent, in text, the type or family which define the intent, method of infection and other properties of the malware which can be of great importance when it comes to researching their behavior. One of the biggest challenges when researching results from multiple AV vendors is, understanding the meaning of their labels. Multiple companies have emerged in last few years where centralized databases are used to collect labeling results from multiple vendors which have the goal of emphasizing the problem of uncorrelated labeling techniques between different AV engines, see [VirusTotal, 2015], [Anubis, 2014]. This means that AV vendors are focusing on detection more than on determining the properties of the malware, leading to false positives as seen in Figure 1.1, where AV vendors tend to detect clean software as malicious programs. The study has been done by [AV-TEST, 2015] on 410 new released samples.

This report will attempt to provide a solution to the problem described by means of another different dynamic approach, without relying on the correctness of the static analysis provided by a single AV

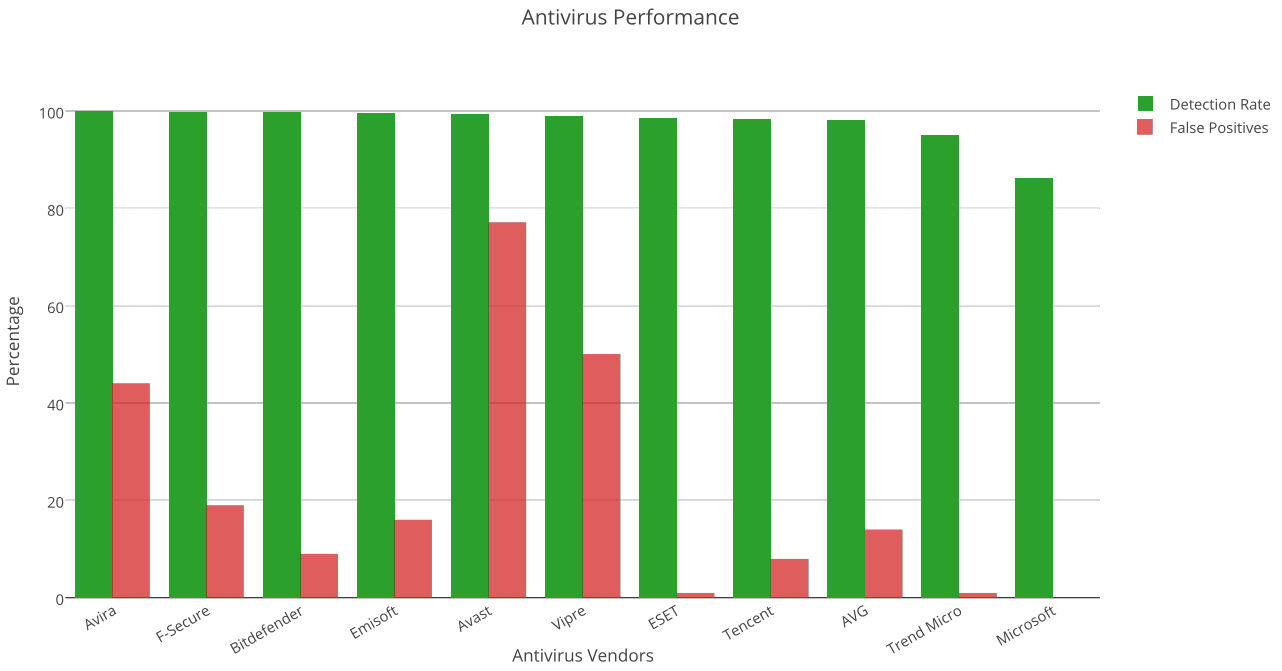


Figure 1.1: Performance of AV Vendors.

program, but instead trying to determine a link between the two methods. Even though the previous research has shown that behavioral similarities of each type can be seen and detected, the results mainly relied on the accuracy of the static analysis provided by a single AV vendor. As code obfuscation and polymorphism are common techniques for hiding the original code of the malicious software, the accuracy of the static analysis will not provide the necessary reliability in order to perform accurate malware detection based on the labels provided by the AV programs. This can lead to false static analysis classifications and therefore will affect the results of the classification when dealing with behavioral analysis.

Next chapter will summarize the work done for the previous report, presenting, in short, the system in use followed by the Preliminary Analysis and Problem Statement. The Technical Analysis provides methods of tackling the problem followed by the System Design and Implementation that describes how the system is implemented provided the information and methods presented in the aforementioned sections.

Chapter 2

Problem Isolation

This chapter contains detailed information, constraints and limitations to form a potentially improved solution to the previous report using the information gathered from novel research. To better understand the basis of this project, a short summary of the previous work will be presented to bring the reader up to speed for the changes that will be introduced. This will include the previous problem statement, data collection, evaluation, analysis approaches taken along with the results gathered. Furthermore, it will be explained how AV companies label and detect malware presenting in detail new approaches of analyzing malicious software behavior.

Finally, this chapter will include a motivation section that will be based on the problem isolation leading to the problem statement. By the end of this chapter the following questions will be answered.

- What were the drawbacks of the previous work ?
- How to correctly capture the family and type ground truth using only behavioral information from malware samples?

2.1 Summary of previous work

The motivation of our previous work, [Pirscoveanu et al., 2015], relied on classifying a large number of increasing malware that emerge each day. In order to be able to keep up with the exponential increase, a dynamic and automated approach was selected to work along side the static analysis. In short, static analysis refers to reverse engineering the malicious code and finding patterns, while dynamic analysis focuses on the behavioral information of a malicious program during its execution in a safe and contained environment. After comparing the two approaches it was decided that dynamic analysis will provide the automation and distribution capabilities needed to analyze the new malware at a rate close to the increasing number.

A system has been designed that is able to collect, analyze and classify a large number of samples in a short period of time using Cuckoo Sandbox controlling a fairly large number of Virtual Machines while storing information in the Not Only Structured Query Language (NoSQL) Database Management System (DBMS) MongoDB. In order to store the unstructured collected data, a non-relation database was chosen to fulfill this task. Predicting the structure of the malware behavioral data can be a very time consuming and complex operation, thus relational databases would have been an unacceptable choice when the structure is not known. Variables like Application Programming Interface (API) calls, created and accessed Registry keys, files and mutexes were chosen as features that should capture the behavioral profile of the malicious software. The features were then used with a classification algorithm that would classify malware samples to their appropriate type. Different representations of the features were created:

- Binary and Frequency representations of API calls.
- Counters of Files, Mutexes and Registry Keys.
- Sequence representation of API calls.
- Combinations between Frequency, Binary and Counters.

2.1.1 Methods

A unique technique of choosing the appropriate features based on their relevance has been depicted to help reduce the large dimensions of the data. The relevance has been calculated by normalizing the variance of the feature values evaluated on the whole sample set. This method was intended to keep the features that better discriminate the malware types and discard those which are redundant or provide no performance gain.

By carefully analyzing multiple supervised algorithms based on their performance on similar data sets, it has been decided that Random Forests (RF) will be used along with labels from Avast. RF represents a classification algorithm that constructs multiple decision trees and performs a majority vote on the decision of every tree. Selected labels have been chosen based on the AVs that had the best detection rate at the date of the scans along with a consistent labeling technique. Avast provided the project with approximately 40,000 samples spread across four malware types consisting of Trojan, Adware, Potentially Unwanted Program (PUP) and Rootkit.

2.1.2 Results

After using RF Machine Learning (ML) algorithm representing a supervised method with labels provided by Avast, it has been seen that the Combination between Frequency, Binary and Counters representations, containing concatenated information, yielded the best results in terms of Precision, F1-Measure and Area Under the Curve (AUC) values. These metrics represent the Receiver Operator Characteristics (ROC), providing valuable information about the performance of the used classifier.

During analysis of the four most seen types consisting of Trojan, Adware, Rootkit and PUP, it has been concluded that, even though high performance has been achieved, a better uniform distribution of the samples should be used to reflect a fair performance evaluation. The Trojan samples covered over 70 % of the samples, which meant that the classifier would probably classify a test malware as Trojan if it does not hold any of the obvious features of Adware, Rootkit or PUP. The published paper is available in the last pages of this report.

Even though the study has shown that the classifier had a fair predictive performance for the tested data-set using AV labeled types, it has to be emphasized that the results are relative to the correctness of the labels. As there is no proof of a link between the labeled information and the malware behavior data, a different approach must be used in order to determine if similarities exist between the two, in order for the classification to be valid.

2.2 AntiVirus Programs

This section presents a short history of AV programs, along with their different methods of detection and labeling. By the end of this section the downsides of using AV labels for research will be emphasized, which by the end of the Chapter will represent the motivation of this project.

AV software have been around since early 1990s when the first malware has been identified. Since then, as Internet access has expanded, malware infections have increased considerably and so did the AV industry. Today there exist approximately 60 different AV programs that detect malware using different methods [Wikipedia, 2015a]. This has led to inconsistency in labeling the detected malware and in some cases in a large number of false positives as pointed out in [Mohaisen and Alrawi, 2014].

With this boom in the AV industry, different companies like AV-Test or AV-Comparatives have been trying to provide crucial information to users by assessing AV performance on a "real-world" virus database, see [AV-Comparatives, 2015] and [AV-TEST, 2015]. Statistics like detection rate and False Positive Rate (FPR), also seen in Figure 1.1 (p. 4) hold precious information about the quality of an AV program. As an example, if a user relies on an AV program that has a low detection rate, the machine he uses will be more prone to infection as the AV is unable to detect all malicious programs. Or, if the AV has a high FPR, the user will receive false alarms about programs that are not affiliated in any way with malicious activity. The discrepancies in the results of different AV vendors can be described by the different methods of detection in use and their appropriate algorithm sensitivity used for finding similarities between samples. These methods are described in the following subsections.

2.2.1 Detection

Malicious software has advanced in both complexity of coding and methods of obfuscation and polymorphism making the detection more and more complex in order to cover all possible use cases. As technology evolved, the detection methods that AV vendors used have changed as well, improving their accuracy for new-released malware. Methods currently used are:

- Signature Based - Most commonly used by AV vendors to fast detect known malicious software. There exist two different signature-based methods:
  - Hash Signatures are used by AV vendors to compare the hash of the file with known malware hashed signatures. This provides a fast detection rate only if the malware exists in the AV database or if the malware has not changed. AV vendors commonly use MD5 message-digest algorithm that provides a very accurate result. For example if a single character is changed in a file-name, the MD5 will return a totally different hash, see [Griffin et al., 2009].
  - Byte Signatures represent a sequence of bytes that are present in executable files or data streams. This method can accurately detect malware families by analyzing sequences in data streams of an executable file [Kephart and Arnold, 1994].
- Heuristic Based - The method attempts to detect malware by simulating the run-time of the suspected executable and determine its intent. This approach is useful when, for example, an updated

version of a known malware is released and it is mostly used alongside signature based detection to form a completed real-time protection in terms of known and new malware threats.

- Behavioral Based - A more advanced method of detection that requires the malware to run and infect a machine in order to record its actions. Behavioral based detection is done in a confined environment to prevent the spread of the malware. Information collected is usually used for classification or clustering of malware behavior [Bayer et al., 2009].

Modern AV programs use all of the above methods to provide a more complete solution to the customer for a higher and more accurate detection of malware. AV vendors do not only detect malware but also label them according to patterns seen when reverse engineering the executable. The following subsection will describe the labeling techniques used by AV vendors.

2.2.2 Labeling

From an AV user point of view, the labels of a detected malicious sample are just names assigned to dangerous software that may harm the integrity of the machine he uses. However from a researcher point of view the labels contain crucial information about the malicious sample that can help solve problems of detection using different methods. The labels provide information about the type, family, version, method of detection and platform of the malware sample. AV companies mostly use their own naming convention of detected malicious code. An attempt to make a common naming technique was made by a company named CARO, called the CARO Naming Convention, see [FitzGerald, 2002], however only a few AV vendors make use of it. Following paragraphs will describe the types and families present in the generated AV labels.

Malware Types

Over the years multiple malware types have been seen executing different malicious actions. From simply presenting the user unwanted content to completely taking over the machine and restricting access to it. The known and most commonly seen malware types are:

Trojan Horse is presented as software that the user might find useful, just like any other legitimate program. By opening the package, this malware releases other types of malware that will infect the machine, including key-loggers, account stealers etc. Compared with Viruses and Worms, Trojans do not replicate on their own but instead they require user interaction to do so. For this reason, this type is one of the most dangerous out there as it is usually detected when it has already infected the machine [Cisco, 2015].

Virus represents a malware type that can exhibit actions ranging from just showing random errors to taking the system in a Denial of Service (DoS) state. The main difference between a Trojan and a Virus represents the ability to self-replicate by becoming part of other legitimate software. These types are commonly spread by sharing files, disks or e-mails to which the virus has attached on [Cisco, 2015].

Adware represents one of the least dangerous types as its only purpose is to display ads to the user. In order to provide the infected machine with ads that the user might be interested in, it logs information like browser history, search engines history or history of installed programs. Depending of the severity of the logging, Adware may be labeled by AV vendors as Spyware.

Spyware represents a type of malware which installs itself without the permission of the user. Used to collect browsing history and tracking information it usually bundles with free software [Symantec, 2009]. AV vendors also name this type, PUP, just because of the bundling with freeware.

Worm represents a similar type as a Virus, being able to do the same amount of damage to an infected machine. The main difference is represented by its independence from other software as it does not require a host program to attach itself to. A worm usually infects its target via exploits or vulnerabilities and it uses different transport protocols to spread and infect other machines [Cisco, 2015].

Bot represents a malware type that grants access of the infected machine to its master. This type can spread using Backdoors opened on the target by a Virus or a Worm and it is mostly known for using Internet Relay Chat (IRC) to communicate with its master. With multiple bots, Distributed Denial of Service (DDoS) attacks can be initiated that could block the services of the target by overwhelming it with requests.

Ransomware represents a more sparse type of malware that takes control of the graphical interface and blocks the user from accessing its machine until a certain amount of money is paid. Most commonly, these types infect their targets via Trojan Horse.

As can be seen from previous paragraphs, different types may exhibit different tasks. A correct labeling of detected malware provides great insight about the methods of removing a malware from an already infected machine. For example if a Trojan-Downloader has been detected, it is not enough to remove it from the system and define the system as clean. It is crucial to look for other types of malware that this type may have injected on the machine in order to make sure that the system is in fact malware-free. This motivates the idea of correct labeling when detecting types to more accurately prevent or neutralize infections. Thus, it is required to have a conventional naming scheme to generalize type labeling over all AV vendors.

Malware Families

Malware Families often represent code similarities that might point to the same source code. As there are infinite number of coding styles and ways of achieving the same goal, finding patterns between malware with either Signatures (Hash or Byte) and Behavioral Analysis most probably denotes that the malware is using the same source code as previously detected samples. Similarities can increase the detection rate as it provides AV vendors with valuable signatures that may or may not change with an updated version of the same malware. Examples of some of the well known malware families are:

Zeus represents a Trojan-Bot which tries to steal confidential information like bank accounts and can also contact a Command and Control (C&C) server. Most used method of collecting information was by web-injecting fields which were asking for sensitive information. This particular malware was distributed via spam e-mails and the number of infections, in 2010, reached millions [Symantec, 2014]. Numerous replicas have been seen and detected as it is using a limited set of file-names like *sdra64.exe* or *pdfupd.exe*.

Hupigon is a malware family rated as critical by TrendMicro, see [TrendMicro, 2015]. This family consists of backdoors or information stealing malicious software. It has been named accordingly to the file names it creates on the infected machine. Variants may use the same pattern of naming files, thus leading to the conclusion that they use the same source code.

2.2.3 Downsides

After describing some of malware types and families and understanding the risk they impose, a short description of the downsides of using AV labels for providing the ground-truth on malware samples will be presented.

For an AV user, a better AV program is described by a high detection rate and not by how correct are the malicious samples labeled. However the research community makes use of the labels provided by AV vendors and require their results to be as close as possible to the ground truth in order to successfully use them in behavioral classification. The paper [Mohaisen and Alrawi, 2014] raises the concerns of inconsistency and evaluates AV vendors in detail, based on four main characteristics:

- Completeness determines the detection rate of each AV vendor based on a test sample set.
- Correctness determines the correctness of the labels based on the types and family definitions.
- Consistency determines the level of similarity between labels provided by different AV vendors.
- Coverage is used to calculate how many AV vendors need to be used in order to maximize the Completeness, Correctness and Consistency.

All of these metrics and their importance will be described in detail in the Technical Analysis part of the report where AV vendors will be tested on the sample set provided by this project.

2.3 Machine Learning

Machine Learning represents the exploration of algorithms that learn from a data set. It is mainly used in conjunction with Data Mining where Big Data can be automatically analyzed by a computer in a short period of time. Using mathematics and statistics, ML algorithms are able to predict how future data will look like given a set of known data.

In malware analysis terms, this discipline can be used to analyze a large amount of malware to keep up with the growing amount of samples and infections each day. It can handle multiple amount of dimensions and it finds patters in the information provided, being able to classify and cluster. These techniques are also known as supervised and respectively unsupervised learning.

2.3.1 Implications of previous work

Previous research exists in classifying and clustering malware behavior using various Machine Learning algorithms in combination with different features and feature representations.

Previous work, see [Pirscoveanu et al., 2015], has mainly relied on supervised learning, using labels from AV program Avast as the ground truth. This made it possible to configure the Random Forests algorithm to learn the behavior of the malicious program using its labels in order to classify newly inserted malware and match the learned behavioral patterns and similarities based on Windows API calls. [Hou et al., 2015] try a different approach of clustering malware behavior using a cluster-oriented ensemble classifier on a data-set collected from Comodo Cloud Security. The features selected were also based on general API calls and the results were satisfactory, obtaining a 96% detection accuracy, outperforming the signature-based detection techniques.

Overall there exist different approaches in classifying or clustering malware behavior using Machine Learning. The choice between the two methods is based on the data-set at hand and at the same time if correct classes are available.

2.3.2 Supervised versus Unsupervised learning

Supervised Machine Learning represents the task of learning from a labeled sample set. Usually the algorithms are split into two main phases called: **training** and **testing**. The training phase represents the learning part of the algorithm where each sample has a desired output. The algorithm learns from the known data and it is then evaluated on the testing set, see [Kotsiantis et al., 2007].

Unsupervised Machine Learning represents the task of finding hidden groups in an unlabeled sample set. As opposed to the Supervised ML, there is no method of evaluating the results of the solution provided by the algorithm, see [Gentleman and Carey, 2008].

When comparing the two methods of learning combined with the discrepancy in AV labels, a problem arises in terms of the correctness of learning in behavioral malware analysis. As different AV programs have different methods and techniques of detecting and labeling malware, using supervised machine learning will yield results based on the labels provided which in the end can lead to an erroneous classification, [Mohaisen and Alrawi, 2014]. On the other hand, unsupervised learning focuses only on the data set and does not need any kind of labels or classes. Data provided by the features will be used to determine the differences between clusters, and in the case of malware, the difference between their behavior.

2.4 Motivation and Limitation

Motivation of this project relies on the discrepancies of obtaining the ground truth from labels provided by AV vendors. It is believed that a better representation and discrimination of malware types and families can be achieved using unsupervised learning without making use of classes. The resulting clusters can then be labeled by analyzing malware samples that belong to a specific cluster or it can simply represent a majority vote from multiple AV vendors. When doing so, the ML algorithm will generate clusters based only on the provided data-set and on the properties it holds. Furthermore a comparison between the created clusters and the number of distinct types or families extracted from AV vendors can be used for evaluation to determine if a link exists between the two methods.

Additionally, supervised learning can be used along with the created clusters to classify new malware in the corresponding cluster. The labels in use will be the ones generated by the clusters and not by AV vendors, ensuring that the labels represent the behavior and not the other way around. The model can then be regenerated every time a new sample is added, after a set period of time or after a set number of new samples. This approach will ensure that the classification will always be run on the most recent data available, providing the highest accuracy possible.

In order to analyze a large amount of samples, for this project it has been chosen to use Dynamic Analysis collected from a Virtual Machine environment. The information will then be processed and passed to an unsupervised ML algorithm that can accurately cluster malware behavior and compare the results with labels extracted from multiple AV vendors.

The chapters that follow present a more technical approach of the topics listed below:

- Dynamic Analysis.
- AntiVirus labels.
- Unsupervised Machine Learning.

This chapter has presented a short description of previous work regarding classification of malware behavior along with the problems encountered. It has also described how AV vendors label detected malicious software along with challenges and potential solutions. A short discussion uncovered that unsupervised machine learning performed on information collected from dynamic analysis of malware can form a potential solution of accurately clustering malicious software types or families. The next chapter provides the problem statement for this project together with additional questions that need to be answered in the Technical Analysis Part of the report.

Chapter 3

Problem

Based on information collected from Chapter 2 along with the motivation and limitation presented in Section 2.4 (p. 12) it has been decided that Clustering can be a potential solution to correctly grouping malware behavior using information collected by dynamic analysis. It is believed that the future system can accurately extract and define the behavior of malware without making use of AV generated labels. In order to find a solution to this problem the following problem statement must be tackled:

3.1 Problem Statement

How to cluster malware in order to discover similarities, using Machine Learning applied on behavioral data generated using dynamic analysis?

3.1.1 Sub-problems

The following questions need to be answered in order to solve the problem statement:

1. Which features should be used?
2. Which feature representation should be used?
3. Which method of defining the number of clusters should be used ?
4. Which unsupervised ML algorithm should be used for clustering?
5. How to perform majority vote on unstructured AV labels ?
6. What kind of information will the clusters hold?
7. How can the clustering results be evaluated?
8. How will the unsupervised results be interpreted in comparison with the supervised results obtained in previous project?

Next part will present the Design Rationale and Methods where the answer to the above questions will be depicted. By the end of the part, an overview of the system will be presented, constrained by answers to the sub-problems, which should solve the problem statement.

INTENTIONALLY LEFT BLANK

Part II

Technical Analysis

INTENTIONALLY LEFT BLANK

Chapter 4

Design Rationale

This chapter provides answers to the different sub-problems illustrated in Section 3.1.1 (p. 13), leading to a finalized version of the system design. In the beginning of this chapter, the physical architecture is described based on the knowledge of the previous project and the improvements discussed in Chapter 2 and in Chapter 3. As no technical details have been discussed so far, the system architecture will be based on the ideas specified in the previous chapters. This chapter is addressing potential solutions and improvements to the previous project in terms of:

- Dynamic malware analysis using multiple Virtual Machines and created Data Structure.
- Malware samples.
- Selection of labels for evaluation.
- Features used to create the malware behavioral profile.
- Feature Extraction.
- Feature Representation.

A final system architecture is presented in the end of this chapter in Section 5.6 (p. 59).

4.1 Technical Overview of Previous Project

In order to better understand the choices that will be made in this chapter, a technical overview of the previous project is presented to form the basis of the overall system.

The dynamic approach of analyzing malware required the system to have multiple secure and confined environments that could host malware samples and capture their behavior. Assessing the performance of the available environments a choice has been made to use Virtual Machines (VM) as they provided fast state recovery, APIs to automatically control the guests and the customization support required by the project. A client-server model has been used to construct the system architecture where a central server would control the state of multiple Virtual Environments over a secure internal network. In order to prevent the injected malware to access the Internet, InetSim was used to created dummy servers that would emulate the most common protocols like : HTTP, HTTPS, DNS, IRC.

Furthermore, Cuckoo Sandbox has been used to control the state of the VMs and inject malware for analysis. Distributon of the analysis setup has been achieved by modifying the source code of Cuckoo such that it would tunnel VM commands using SSH.

The next section will describe the system architecture used in this project and will present the improvements of choices made in previous work and presented in this section.

4.2 System Architecture

This chapter provides an overview of the system architecture chosen for this project along with the improvements introduced to the previous work, presented in Section 4.1 (p. 17). Figure 4.1 describes in short the architecture of the system:

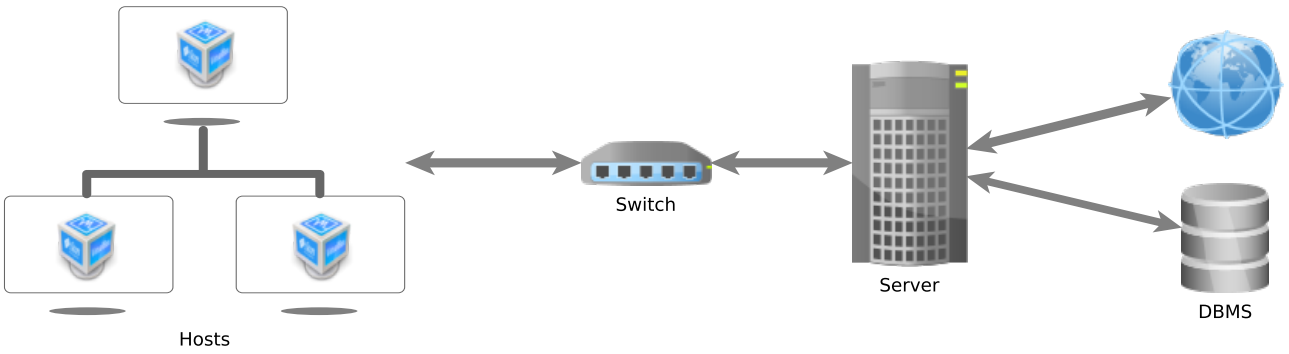


Figure 4.1: Physical system architecture based on knowledge from chapter 2.

The idea of the system has not been changed in comparison with the previous project where a centralized server controlled the injection of malware samples in multiple Virtual Machines using a secure internal network. A number of improvements, in terms of hardware, have been introduced to be able to cope with a larger number of malicious samples.

Some of the limitations observed in previous runs have affected and limited the performance of the analysis machines in use. The hardware that imposed limitations are:

1. Random Access Memory (RAM) provided a bottleneck in terms of the number of Virtual Machines a Host can sustain. As the operating system in use, Windows 7, required a minimum of 512 MegaBytes (MB), the host could only sustain 4 machines at a time.
2. Hard-Drive Disk (HDD) provided a bottleneck in terms of both throughput and storage for the hosts and also for the main analysis and storage machine.

In order to collect and analyze a set amount of samples in a short period of time, certain upgrades to the analysis setup were necessary to compensate for the larger number. The hosts have been installed on two Redundant Array of Independent Disks (RAID) 0 disks. The number of hosts has been increased to ten and the number of Virtual Machines used on each host now varies from four to eight, summing up to a total of 30 confined environments. The main analysis machine has also been introduced to three 750 GigaBytes (GB) RAID 0 disks for the database partition and three 2 TeraBytes (TB) RAID 0 disks for the raw analysis data, providing faster access at the cost of redundancy.

As the number of components used in the system is fairly large, the following list provides the hardware and software used based on decisions made in the Problem Isolation as well as decisions made in previous project.

- 10 machines in use.
  - 9 hosts:
    - Disks : Two Raid 0 HDDs.
    - Operating System : Ubuntu LTS 12.04
    - Virtual Environment : VirtualBox 4.3
    - Two to eight guest Virtual Machines installed on each host.
  - 1 server:
    - Storage Disk : 4TB consisting of two 2TB Raid 0 HDDs.
    - Database Disk : 2TB consisting of three 750GB Raid 0 HDDs.
    - Operating System : Ubuntu LTS 14.04
    - Database Management System : MongoDB v3.0.1
    - Malware Analysis Tool : Modified Cuckoo Sandbox v1.2
    - InetSIM:
      - Iternet Emulation software.
      - Custom configuration to respond with a default response based on requested protocol.
      - DNS requests to microsoft.com return the correct IP.
      - HTTP requests to ncsi.microsoft.com return correct response to make the OS believe it is online.
- 30 guests:
  - Operating System : Windows 7 Service Pack (SP) 1, Danish Version
  - Disabled Firewall
  - Disabled Microsoft Anti Virus
  - Disabled UAC
  - Pre-installed common programs from `ninite.com`
- Two Gigabit switches connecting the hosts with the server.
- 270,000 malware samples
  - $\approx$ 190,000 new samples
  - $\approx$ 80,000 reused malware samples from previous project, retrieved from `virusshare.com`

This section has presented an overview of the software in use along with the physical architecture of the analysis setup. The following section will explain the Data Structure used, presenting the chosen Malware Analysis Tool as well as the Database Management System.

4.3 Data Structures

This section provides a short overview about the Analysis program used along with improvements made to the data structure it provides.

4.3.1 Cuckoo Sandbox

Cuckoo Sandbox has been the open-source analysis program chosen in the last project as it has provided the control interface for multiple VMs and the ability to legally customize its code to fulfill the scalability needs. This project has continued using the software, updating it to version 1.2, containing improvements in both performance and compression of data. After the update, the modifications done in last project had to be reapplied, in order to control Virtual Machines over the network, by tunneling SSH commands.

Cuckoo Sandbox is responsible for injecting malicious software to a clean Virtual Environment and collect its behavioral information, saving it for further use. After the analysis is finished, it is responsible for restoring the infected machine to a previous clean state in order to be used for another infection with a new sample. It has to be mentioned that, due to hardware space limitations, the malware is limited to execute for at most 200 seconds after which it is killed and the analysis processes is terminated. However, the execution time of each malware differs from a few seconds to more than three minutes. There are three main reasons why the analysis process could terminate before schedule:

1. The sample detects that it is being monitored and it terminates to prevent analysis. A detailed report on malware evasion techniques in Virtual Environments has been done by [Wueest, 2014] where the results have shown that approximately 18% of malware detect the presence of a VM.
2. The injected malware might have a trigger-based behavior denoting that it will only execute if certain conditions are met. Some malware may execute only at given dates or times, when the infected machine is idle or if Internet connection is available, see [Brumley et al., 2008]. Other examples are also related to human interaction where the malware will wait for mouse or keyboard input before executing its intended actions, see [Wueest, 2014].
3. The executable in which the malcode is packed is not compatible with the OS installed on the guest machine.

For the examples provided in items one and two of the above enumeration, an assumption is made in this project that such malware will not execute more than 50 API calls resulting in a partial behavioral profile. Example number three on the other hand represents a failed analysis where the malware was unable to execute on the target machine due to incompatibility issues. Such a sample, with a failed analysis, is always marked by Cuckoo with an error message implying that the analysis is incomplete and should not be used for research. It should be noted that there exist a number of differences between the given examples. As the first two would execute and create only a single process denoted by the executable and at the same time have a small amount of API calls, the last example will create no processes and no behavioral information will be available.

Next subsection will explain the modifications made to the output of Cuckoo in order to keep only relevant information that will be used for this project to improve performance and reduce the analysis time.

4.3.2 JSON Output

The output of Cuckoo is represented using one JavaScript Object Notation (JSON) file for each tested malware sample. The JavaScript Object Notation represents a light-weight format that can contain nested structures combining Strings, Integers, Booleans and single or multidimensional arrays. The values are stored using name-value pairs that can contain universal data structures used by most programming languages, see [Crockford, 2013]. Cuckoo uses this format to output the behavioral data in a more structured way that is at the same time easy to access and easy to understand. The output also contains static analysis data, network traces, file information and Virus Total reports containing detection results for approximately 57 AV vendors, [VirusTotal, 2015].

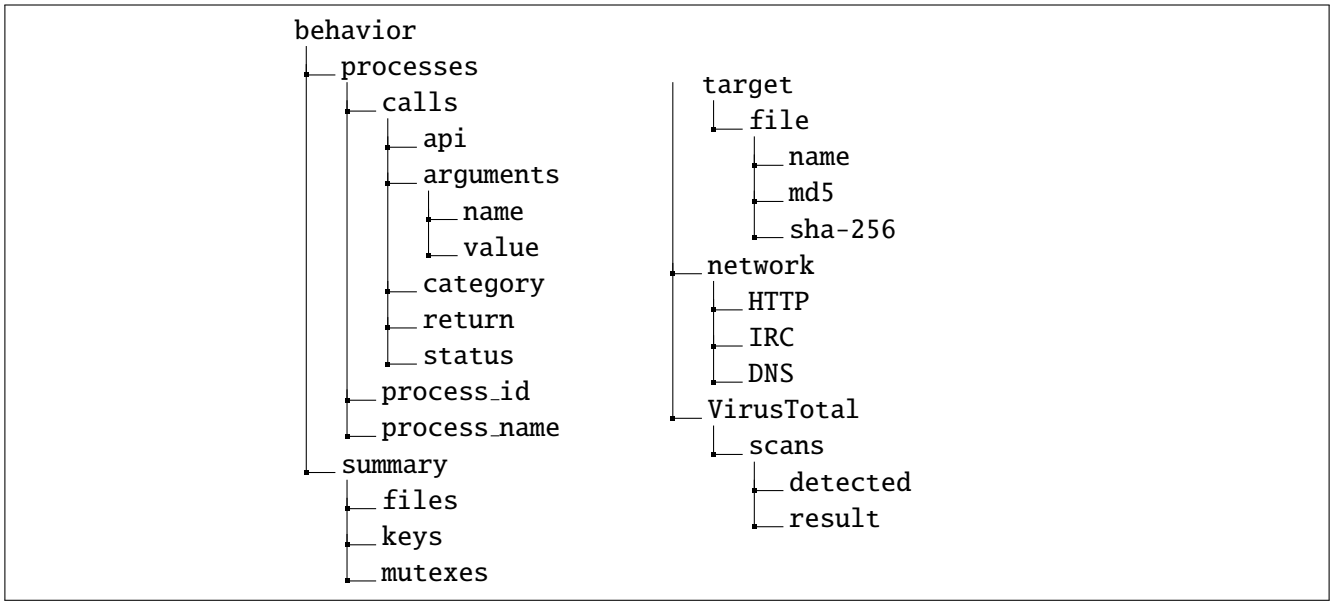


Figure 4.2: JSON output of Cuckoo

Figure 4.2 presents a directory tree-like diagram that describes a sub-set of values present in the structure of each JSON file provided by Cuckoo. As it is supposed to be unstructured, some information may or may not be present. Furthermore, the structure is complex and contains multiple nested values. For example in order to access the *name* and *value* arguments of an API call, four different lists need to be opened for each API of each process of each sample. This provides a great loss in terms of performance and time, as iterating through the database in previous project took approximately three hours. In order to improve access times, the structure of the database has been rethought. Removing irrelevant information like timestamps, various target information related to Cuckoo and information related to static analysis, the structure has been reduced to the structure seen in Figure 4.3 (p. 22).

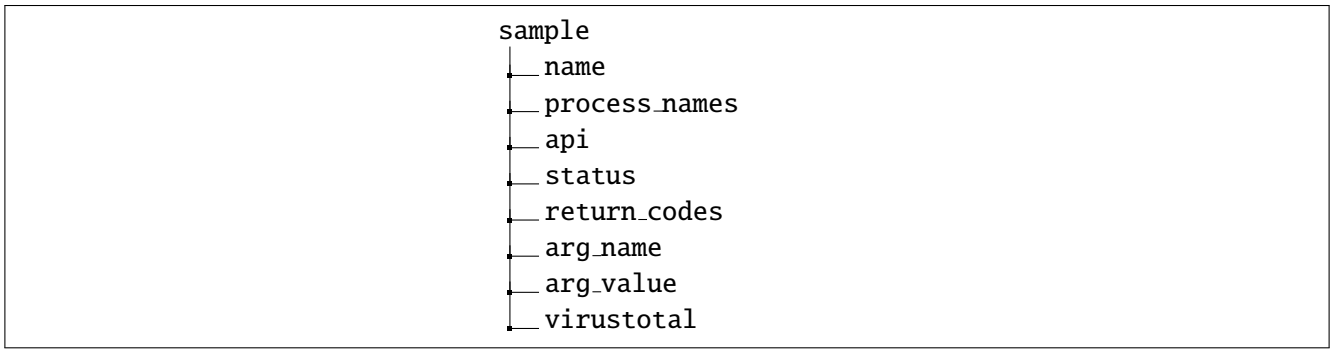


Figure 4.3: End result of JSON modificaton

Detailing the diagram in Figure 4.3, the JSON files now contain the following information:

- `Name` object represents the file-name of the injected malware in the analysis system. It is mainly used to keep track of the samples and extracted data. In some cases, the name also contains information about the type or family of the malware. This information is now top-level and can be accessed faster.
- The object `process_names` contains a list of names denoting the created processes during the analysis. These processes names are only related to the injected malware and are used to keep track of the source of the API calls.
- The `api` object represents a list of API calls initiated by the malware. As timestamps and sequence ids have been removed from the original objects, the calls are ordered in the correct sequence of calling. It has to be mentioned that the number of items present in `process_names`, denotes the number of lists present in this object.
- `status` object represents a list of boolean values, `true` or `false`, keeping track of the status returns of API calls. Each value in this object corresponds to the API call at the same position in the `api` object.
- `return_codes` object represents a list in close relation to the `status` object. If a True value is seen as the status of a specific API, then the corresponding return code will be 0 represented by a 32bit hex-decimal. If a False value is seen, the the return code will vary depending on the error. These error codes can be converted to a human readable form using the Microsoft Developer page, see [Microsoft, 2015a].
- The `arg_name` and `arg_val` objects are also correlated, sharing the same position in their list. They provide passed information to a certain API, and represent the requests. From the value, information like files, mutexes, DNS requests, registry keys can be extracted.
- The `VirusTotal` object contains detection results from 57 Anti-Virus programs along with their appropriate labels. As compared with last project, where a limitation of Service API requests prevented gathering label information for a good portion of samples, for this project, VirusTotal has provided an unlimited academic API to fulfill the requirements of a large data-set.

Using this compressed JSON structure has allowed the project to advance at a faster rate, allowing extraction of data in a matter of minutes instead of hours. Next subsection will shortly describe the Database Management System that will store the JSON representations for fast retrieval.

4.3.3 MongoDB

MongoDB has been the choice of DBMS for this project. As it can handle direct JSON inputs, it represents a fast solution for storing the behavioral information provided by Cuckoo. In comparison with the previous project, the version has been updated to 3.0, containing improvements in both performance and compression of data. MongoDB represents a unstructured query language database that is able to save information in a binary format called Binary Script Object Notation (BSON), and perform complex operations consisting of matches, projections, aggregations, etc [Chodorow and Dirolf, 2010].

This section has presented the Data Structures used in the project along with a short description of the features that can be extracted using Cuckoo. The next section presents the malware samples that will be injected to Cuckoo and stored in MongoDB for further analysis.

4.4 Malware Samples

This section will present a technical overview of the malware samples in use, types and families present in the database and the changes introduced to the system to overcome the new challenges in choosing the most appropriate labels.

The amount of samples used for Machine Learning can have a significant impact on the quality of the results providing more diverse information that will eventually discriminate better between types or families. In the previous project 80,000 samples have been used from which only approximately 40,000, spread over 5 types, had enough information to be used in classification. The samples have been gathered from ViruShare, and are available at [VirusShare, 2014].

In this project the amount of samples have increased considerably to approximately 270,000, combining the data-set used in previous project with 190,000 new samples, accounting for as much as 36 types giving the opportunity to cover a more broad spectrum of malware behaviors. The large number of samples could be a possible solution to uniformly gather information from a set amount of samples thus resolving the downside concluded in previous project and described in Section 2.1 (p. 5). The new samples have been gathered from VX Heavens website before it was shut down. The database is still available at [VXHeavens, 2010].

A temporary list of types has been extracted from the labels generated using a unified method of static analysis that came with the samples in use. The labeling has been done by the creators of the malware before submitting them to VX Heavens. The list of types and families, seen in Table 4.1 and Table 4.2, will be used as reference, to select a subset of samples in order to avoid a bottleneck in computational power. As this project represents a student thesis, computational performance is limited by the available hardware.

Type	Samples	Type	Samples
Backdoor	51459	Trojan-Dropper	8193
Trojan	46143	Trojan-Banker	6986
Trojan-Downloader	44410	Worm	6152
Trojan-GameThief	29050	Email-Worm	3315
Virus	26941	Rootkit	3210
Trojan-PSW	16889	Trojan-Clicker	2844
Trojan-Spy	11748		

Table 4.1: Number of samples for 13 most seen types in the sample set.

Family	Samples	Family	Samples
Agent	32012	Banker	5816
OnLineGames	26329	Banload	5255
Hupingon	16472	Zlob	4695
Delf	10024	Obfuscated	4242
Small	8996	Buzus	4010
VB	8517	Autorun	3766
Magnia	7160		

Table 4.2: Number of samples for 13 most seen families in the sample set.

4.4.1 Malware Types and Families

As mentioned in Section 2.2.3 (p. 10), there exists an inconsistency in the labels presented by AV programs. A more detailed and technical explanation will be depicted, and a decision will be made on which labels should be used in order to achieve the best predictive performance when using the data in an ML algorithm. The following subsections will describe in detail the Completeness, Correctness, Consistency and Coverage metrics used to evaluate the AV vendors.

Completeness

The completeness metric refers to the detection rate of each AV. As different methods use specific parameters to detect malicious code, differences between the number of infections detected vary from AV to AV. To illustrate this difference, Figure 4.4 presents the number of detected samples for each AV. It has to be mentioned that the scans have been performed using VirusTotal API on the date of analysis. This information might change with time, as AV vendors update their signatures on a daily basis, thus making it possible to detect more malware [Gashi et al., 2013].

From a malware sample set of approximately 270,000 samples, no AV has a detection rate of 95% or more. Big AV vendors like Symantec and Microsoft fail to detect $\approx$ 80.000 samples , which reflects to 29% of the total samples. With a large number of users that rely on the services provided by these vendors, a detection rate of 71% will leave the users vulnerable to infection from undetected malware. This provides an even stronger incentive to provide a system that can dynamically detect malicious software with a

high detection rate.

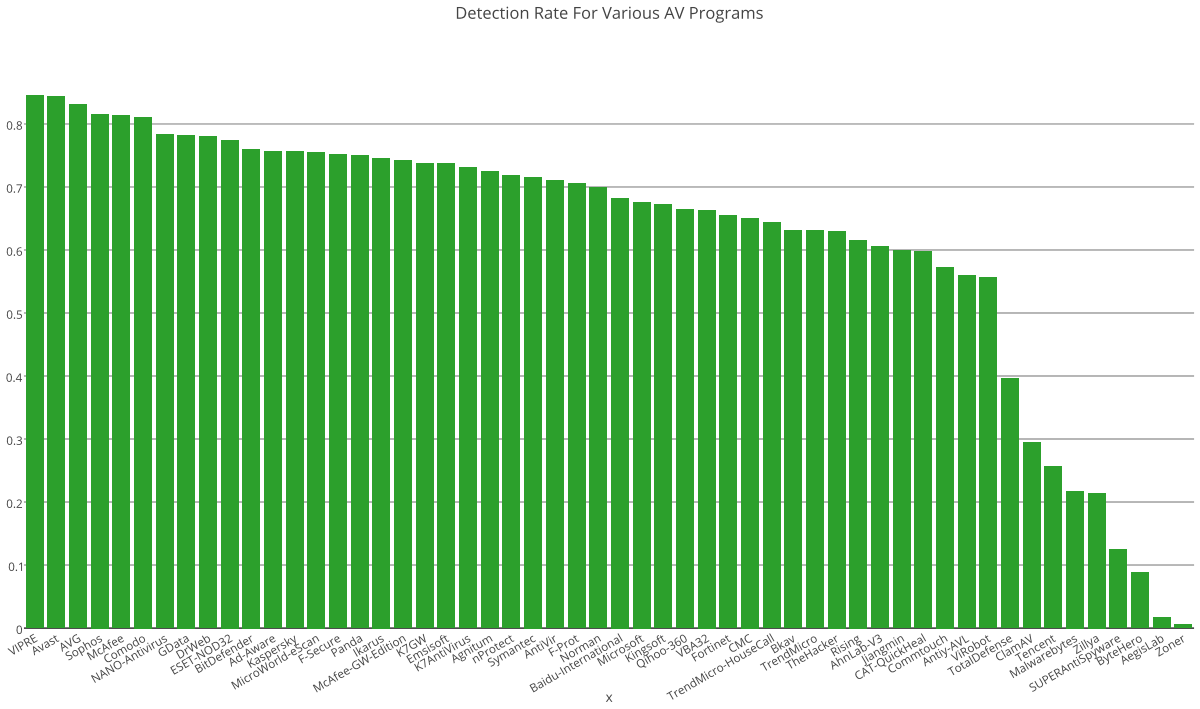


Figure 4.4: Detection rate for tested AVs on the current sample set.

Correctness

The correctness metric describes samples that have been correctly labeled. As this project does not rely on classification, the correctness of the labels is only useful to name the clusters after they have been created and to evaluate the differences between information from labels and the information generated using dynamic analysis. In comparison with the other metrics, this cannot be verified as it requires extensive static analysis on each sample.

Consistency

The consistency metric describes the similarities in labeling between multiple different AV vendors. As with the completeness metric, consistency is mostly defined by the methods used when analyzing the malicious code. To illustrate the differences in labeling, Table 4.3 presents different labels generated by AV vendors on the same sample. To fully understand the extent of the problem, the sample chosen has been detected by 90 % of all AV vendors.

By analyzing Table 4.3, it can be concluded that an automated method of extracting labels of each AV vendor represents a very time consuming task as they mostly use custom naming conventions. Some of the AV vendors, for example Microsoft, follow the CARO naming convention. This naming scheme provides easy access to the information provided by the label, using consistent naming for malware types, families and versions. The CARO scheme is represented by the syntax seen in Code snippet 4.1.

AV	Label
MicroWorld-eScan	Adware.Generic.299810
BitDefender	Adware.Generic.299810
McAfee	Generic.dx!F4AB65EB9C32
Zillya	Dropper.Agent.Win32.116628
K7GW	Trojan (0001140e1)
Symantec	Trojan.Zlob
TotalDefense	Win32/Runbot.A
Avast	Win32:Oliga [Trj]
Kaspersky	Trojan-Dropper.Win32.Agent.bhbp
Microsoft	Trojan:Win32/Adept.B
Ad-Aware	Adware.Generic.299810
Comodo	TrojWare.Win32.TrojanDropper.Agent.527360
VIPRE	Trojan.Crypt.Zcrypt (v)
AntiVir	TR/Drop.Agent.52736
McAfee-GW-Edition	Heuristic.LooksLike.Win32.Suspicious.C
Kingsoft	Win32.Hack.Unknown.(kcloud)
SUPERAntiSpyware	Trojan.Agent/Gen
GData	Adware.Generic.299810
Panda	Generic Trojan
ESET-NOD32	Win32/Agent.NOU
Fortinet	PossibleThreat

Table 4.3: Example of inconsistency in AV labeling on the same sample.

Code snippet 4.1: CARO Naming Convention

```
1 <malware_type>://<platform>/<family_name>.<group_name>
```

where,

- *malware_type* represents the type of the malicious software as presented in Section 2.2.2 (p. 8).
- *platform* represents the targeted machine platform. This could be DOS, Win32, Win64 etc.
- *family_name* represents the family of the malicious software as presented in Section 2.2.2 (p. 9).

If all AV vendors would use such a naming convention, then performance analysis between different vendors can easily be performed. However, such convention is not used and therefore a different approach must be found in order to extract types from multiple AV vendors for further analysis.

With that being said, the main focus is shifted to manipulating the full label provided by each AV. There exist a significant number of algorithms that are able to determine the consistency between words, being able to find patterns and similarities using complex and time consuming calculations between each character and their position in the word. Some of the most common algorithms are:

1. Hamming Distance - calculates the number of substitutions required in order to convert a word to another word. Hamming algorithm requires the two words to have the same length and the consistency between the words varies from one, representing the maximum value thus maximum similarity, to zero representing no similarity, see [Hamming, 1950].
2. Levenshtein Distance - calculates the number of substitutions needed to convert one word to another with no limitation to its length. This algorithm represents a faster approach trading off accuracy for performance, see [Hirschberg, 1997].
3. Ratcliff and Obershelp Algorithm - a more accurate algorithm of calculating the consistency with a trade off in terms of complexity and run-time, see [Ratcliff and Metzener, 1988].

The presented algorithms have been tested on the labels provided by the sample set of 270,000 malware, in terms of complexity, computational speed and their ability of computing string similarity with different lengths. Complexity is defined by how many actions are required to reach the final result and is directly correlated with the Computational Speed. The results after testing the three algorithms can bee seen in Table 4.4. It has to be mentioned that the Compuational Speed is based on the number of samples used for this project and the values may change if a different sample-set is used.

Properties	**Hamming Distance**	**Levenshtein**	**Ratcliff**
Complexity	$O(n)$	$O(n * m)$	$O(n^2)$
Computational Speed	N/A	30m	over 10h
Adaptive Length	✗	✓	✓

Table 4.4: String similarity methods.

It can be denoted that the choice is mainly dependent on the run-time and defined by the number of labels used for comparison. The Levenshtein distance algorithm has a complexity of $O(n * m)$ defined by the length of the two strings, while the Ratcliff algorithm has a complexity of $O(n^2)$. The differences between the two methods can be directly denote in the run-time, where the difference is significant. While Hamming Distance has a linear complexity, it cannot be used to calculate the similarities as it requires strings of the same length. With that being said, Levenshtein distance has been chosen to perform similarity measures between labels.

By looking at Table 4.3, it can be seen that some labels contain parts that have meaning only to the party that has created the labels. This information can be considered irrelevant and does not need to be taken into consideration when calculating the consistency. Thus, the labels are split into tokens taking as delimiter non alphanumeric characters like dots, commas, lines or semicolons. Each token is then compared with other tokens from labels of the same sample, adding the label to a counting vote if it matches with a certain probability. This ensures that every AV is assigned the same weight in the vote. In the end the label with the most votes is selected to be the majority vote for the sample.

Coverage

This metric describes, based on the above presented metrics, how many AV vendors are required to maximize Completeness, Correctness and Consistency in order to provide the most accurate detection rate and labels. The coverage and the decision imposed by this metric can yield different results depending on the sample set in use.

Following the four metrics used to analyze the performance of AV programs a temporary solution is necessary in order to efficiently and correctly evaluate malware types or families. Note the emphasis put on **temporary**, as this will provide a solution for this project and not a general solution that can be used by AV programs. Fulfilling the requirements of the analysis metrics, as concluded in the cited paper [Mohaisen and Alrawi, 2014], the AV companies need to share analysis data and use a unified method of labeling that best describes every malicious sample detected.

In order to provide a temporary solution based on the four presented metrics, it has been decided that the clusters resulting from the ML algorithm selected in this project, will be named by running a majority vote between AV vendors using a tokenized Levanshtein distance algorithm. Furthermore, Microsoft labels will also be used for evaluation as the AV uses an easy to parse naming convention and at the same time it has the lowest False Positive rate in comparison with other vendors, see [AV-TEST, 2015].

This section has presented a technical overview of malware samples in use and the changes introduced to the system to overcome the new challenges in choosing the best labels. Next section will present, in detail, the parameters that can be extracted from Cuckoo providing information of the relevancy to malicious actions.

4.5 Features

This section includes detailed information about the features extracted from the analysis system. Defining common meaning to the information provided by Cuckoo from the collection of behavioral data, a decision will be made on which features will construct a better behavioral profile for each malware based on their meaning.

Cuckoo produces several metrics which provide different information in both meaning (purpose) and level of detail. Features are extracted from MongoDB after it has been populated with the behavioral information using the JSON format presented in Section 4.3.2 (p. 21). The *behavioral* JSON object contains information of the created processes with the corresponding API calls and a summary of uniquely accessed/created/modified files, mutexes and registry keys. More data can be extracted from the network traces which includes Domain Name Service (DNS), HyperText Transfer Protocol (HTTP), IRC, etc.

Next subsections describe each feature to determine their use in defining the malware behavior profile.

4.5.1 API

Microsoft Operating System (OS) provides several ways to interact with the system in order to make interaction and exchanging information a simple task for programs and applications. Even though this makes the task of connecting and using features of the OS easy, it also provides easy access hackers to do so as well by using malicious software to take control or gain sensitive information from the infected machine. The API calls are split in multiple categories, see [Wikipedia, 2015c] and [Microsoft, 2006]:

1. Base Services - interaction capabilities with devices, file system, processes and threads.
2. Advanced Service - interaction and access to Registry, Services, and user accounts.
3. Common Control Library - provides advanced control and interaction capabilities for status bars, progress bars and tabs.
4. User Interface - capability of creating and interacting with windows, buttons and scroll-bars.
5. Network Services - access to creating/modifying and controlling network aspects of the machine.

As the number of malware has increased with the number of Internet connected users, Microsoft has introduced in its latest versions of Windows OS a technology called User Account Control (UAC). This improvement required the Administrator of the machine to approve the run-time of an untrusted software before the OS gives it access to the most sensitive API calls, see [Microsoft, 2015b].

Even though, when introduced, UAC has prevented a large number of malware to execute and spread, this technique requires human interaction. This means that every single user must have knowledge about the usage of UAC in order to prevent infections. A study has shown that approximately 59 % of Windows users will disable their UAC functionality while, out of the ones who don't, 21 % have answered wrong in the UAC prompt thus allowing infection, see [Motiee et al., 2010]. Setting the human factor aside, zero-day malware are able to bypass the the UAC prevention technique. This makes it more and more relevant to detect and prevent malicious code from running on users Personal Computer (PC) by running the code in a confined environment and decide, based on the actions it does, how can it affect the infected machine.

As APIs define the interaction between software and OS, each call must provide additional information about the task it wants to do. This information is passed to the OS via arguments. If, for example, the software does not have enough privileges to run a certain command or task, the API call will fail and will provide the software with a coded response explaining the failed call. In previous project, the API calls have been extracted if they were executed, neglecting the succeeded or failed status returned by the Operating System. Going a level of detail deeper into the problem, in this project, the status of the API call along with the return code provided by the Operating System if the call has failed, will be collected and analyzed carefully. By using the full list of error codes provided by Microsoft, a correlation between a failed API call and the reason of failure will be extracted to possibly construct the behavior profile of the malicious software.

The complete list of the **157** seen API calls during the collection of behavioral data can be found in Appendix C (p. 115). Furthermore the list of seen API calls has been split into two main categories: Passed, see Appendix D (p. 116) and Failed, see Appendix E (p. 117). The **137** Return Codes related to

the Failed API calls can be found in Appendix F (p. 118). It has to be noted that the Return Code of all Passed APIs is denoted by a hexidecimal value of *0x00000000* which has not been added to the list as it would have interfered with the same return code of a Failed API call, denoting a delay imposed by the OS.

4.5.2 Files

Information extracted from Cuckoo related to files accessed/created/modified can be of a great importance to detect and determine the type of the malicious software.

The file extensions or headers, even though customizable, can provide crucial information about the purpose of created or dropped files. A strong relation between the file parameters and API calls exists when library files or Dynamic Link Library (DLL) are imported. These files denote the API calls to which the program that imported them has access to. Similar functionality is also provided by other file extensions like Object Linking and Embedding Control Extension (OCX) and Driver (DRV) files. These file extensions are packed like DLL files however they contain functions for User Interface (UI) and respectively hardware drivers that can be used by the importing software, see [Wikipedia, 2008].

However taking into consideration that the number of possibilities in naming a file is very large, this parameter will most probably not be considered to be part of any features representation regarding type or family clustering. This might be used, as in previous project, as a counter to determine the number of files that have been accessed/created or modified on a per sample basis.

4.5.3 Mutexes

The term Mutual Exclusive (Mutex) object is used as a locking mechanism to share access of system resources, just like a semaphore object. Usually used by legitimate software to control which thread or process accesses the information, it can also be used by malicious software to prevent infection of an already infected system with the same malicious code, see [Wikipedia, 2005a].

For detection and analysis purposes, mutexes can be seen as traces of installed and running software, either clean or malicious. Detection can easily be done based on the naming of the mutexes. The name represents the connection to a specific process thus identification can be done by searching for random and unusual mutant names that represent a high probability of infection. Crucial information can be extracted from mutexes, referred to information accessed by the infected process and is closely related to family detection.

4.5.4 Registry Keys

Registry keys represents the Windows database containing kernel, devices, services, user interface and third party options. Information like System Identification (ID) or third party serial numbers can be extracted via exploits used by malware. Thus, this feature can be characterized as very sensitive data, see [Wikipedia, 2005b]. The format of the registry key is defined as a key-value pair, just like the

JSON format. Different keys have different meanings and can contain other key-value pairs inside. The full list of connotations regarding Registry keys can be found on the Microsoft Developer page, see [Microsoft, 2008d].

A link can be made between the type of the malicious program and the information it accesses or modifies from the registry keys. For example, Ransomware, will modify the registry keys of Microsoft OS user interface to prevent the user from controlling the machine if he does not pay a set amount of money. On the other hand, malware families, can be depicted from the semantics of the registry keys it crates. This means that same families will most probably use same wording or a random sequence of letters of a certain length. This can be detected by Machine Learning algorithms by possibly creating features based on the length of the keys stored.

4.5.5 Internet Traffic

Internet traffic, from a multitude of protocols, can provide malware with ways of controlling the infected machine or was of sending sensitive information to a bad-intended third party.

HTTP protocol is most commonly used by Adware to provide information collected from Registry keys and Cookie type of files. This way it provides information about interests and recently searched terms used to obtain ads that the user of the PC would be interested in. On the other hand, IRC protocol, can be used to provide control of the infected PC to its bot master. Infected machines can then be used as zombies to maybe contribute to a DDoS.

By analyzing the activity of Internet traffic from different kind of protocols it would be possible to determine the type of malware. On the other hand, families can be distinguished from DNS requests, from similarities in the location of the domain registrar or from the country origin of the IP addresses linked with the requested domains. As mentioned before, this represents a more semantic approach that would probably point to the same source code and inevitability to the same family. Cuckoo provides an overview of the DNS requests made by the malware along with the appropriate IP addresses which is done separately from the server.

4.5.6 Choice of Features

This subsection will provide a short conclusion on how the presented Features can contribute at defining the malware type or family.

Overall API Calls, Files, Mutexes and Registry keys provide crucial information to help detect and determine the **type** of the malicious software. This information can be used through out the project to construct features to be used in an unsupervised ML algorithm.

On the other hand Files, Mutexes, Registry keys and DNS requests along with the corresponding IP responses provide crucial information in determining the **family** of the malware using a semantic approach. The features created can be used in an unsupervised ML algorithm to construct and determine

the clusters representing different malware families.

The next section will provide methods for extracting the selected features in order to guarantee the quality of the extracted data.

4.6 Feature Extraction

This section provides an insight about different extraction methods, and the challenges they impose, of features presented in Section 4.5 (p. 28). By the end of the section a decision will be made that will define the set of features to be used in Machine Learning.

Previous work has relied on creating a white list by running a clean or benign file through the analysis setup and record actions of the Operating System. This implied removing any traces of files, mutexes, registry keys or DNS requests from the malware behavioral data that was also seen during the execution of a clean file. As the idea sounded promising and it yielded a large decrease in the number of files, it has been used for previous project. The idea behind Feature Extraction will now follow and trace API calls of each process generated by the malicious software and will ignore any information given by the Virtual Machine that it has been ran on.

There are two main methods of extracting the behavioral data for the injected malware:

1. Snapshot Comparison - this method relies on changes recorded by the Virtual Machine from the moment it has been restored, and the malware samples injected, until the analysis has finished. This implies that files, registry keys that have been modified in any way by any service or process running on the system will be recorded. Such a method presents a lot of noise as the malicious actions are mixed with the clean system actions making it hard to accurately analyze the behavior of the intended malware. Noise can be reduced by running a clean analysis and removing any actions that are common to both the clean and infected system. The clean analysis refers to running the Virtual Machine without injecting any malware and collecting the actions of the Operating System. This information is provided by Cuckoo under the *dropped* and *static* objects.

2. API Traces - this method relies on API calls recorded during the malware execution. API traces cannot be recorded over the whole system, instead it requires a target Process ID (PID) from which it will record the actions. If the target PID forks or creates a new process, the tracer will record its API calls as well. At a first glance, this method provides better information, with less noise, in comparison with the previous method as it can extended to child processes that the main process creates.

In order to chose the best method of extracting and filtering data, some comparative properties are taken into consideration like, **Complexity**, **Computation speed** and **Quality of data**.

Complexity

Complexity refers to the amount of information required to use one of the methods. Snapshot comparison requires no external information and can be achieved only by comparing the behavioral information between the infected and clean state of the Virtual Machine. Filtering can be done by simply creating a regular expression or by iterating through all values in order to remove or keep certain behavioral information. On the other hand, analyzing the API traces requires previous knowledge of each API action, thus resulting in a more complex action to achieve. The complexity of using API traces increases even more with time, as Operating Systems introduce newer API calls with newer versions.

Computational speed

Computational speed refers to the approximate amount of time, or actions, required for the method to achieve the Feature Extraction. Both methods rely on lists that need to be compared with the malicious behavioral and require approximately the same amount of iterations to perform extraction. The computational speed is fairly important in malware analysis as the system aims to analyze a large amount of samples in a short period of time. Increasing the amount of iterations will drastically increase the extraction time for large amount of malware samples.

Quality of data

Quality of data refers to the correctness of the collected data. Quality of the data is assumed to be high where noisy information is not present and it is assumed to be low otherwise. The Snapshot comparison method cannot guarantee the quality of data as certain system operations are running at a certain interval or time. This means that noisy information is bound to be present in this method of Feature Extraction. On the other hand, information extracted using API traces relies on carefully extracting files, mutexes, registry keys from the parameters and arguments of the appropriate API call. As the API traces are available for the malicious process itself and its childs, the information collected is guaranteed to not have any noise from the Operating System actions.

Requirements	Snapshot Comparison	API Traces
Complexity	✓	✗
Computational Speed	✓	✓
Quality of data	✗	✓

Table 4.5: Filtering methods comparison

Table 4.5 presents the pros and cons of the extraction methods, shown in a more compact and understandable way. The API trace method provides an increase in Quality of data without increasing the Computation Speed but at the cost of Complexity. This provides an incentive to use this method instead of the previously used method with white-listing.

4.6.1 Extracting Files

Filtering files from API traces requires knowledge about API calls that provide developers means of accessing files on the machine. The full list of API calls that meet this requirement can be found on Microsoft Developer website, under File Management Functions, see [Microsoft, 2008a].

As explained in the API section of the report, each call requires at least one input argument to provide information like location, name and privileges. These arguments can be extracted from the API call, allowing the collection of files that have been accessed, created, modified or deleted. Running filtering based on the information presented has yielded the following results, as seen in Figure 4.5. The API calls *OpenFile*, *OpenFileEx*, *CreateFile*, *CopyFileEx* have been explicitly used in combination with the *FileName* argument to extract the name of the files. The name of the files can be found on the x axis, while on the y axis the occurrence of that particular file is presented over all samples. Denoted with red are the occurrences of files filtered with API traces, while with green are represented the occurrences of files filtered with Snapshot Comparison.

Figure 4.5: Bar chart of 20 most seen files using both methods.

4.6.2 Extracting Mutexes

Filtering mutexes requires information about the API calls that allow management of mutants. The full list of such API calls can be found on the Microsoft Developer website, see [Microsoft, 2008b]

As with the files, the API call for creating and controlling a mutant require input arguments from which the name can be extracted. Running filtering based on the information presented yielded the following results, as seen in Figure 4.6. The API calls *OpenMutex*, *CreateMutex* and *CreateMutexEx* have been

explicitly used in combination with the *Name* argument to extract the name of the mutex. The name of the mutex can be found on the x axis, while the y axis holds the occurrences of a particular mutex over all samples. To denote the differences between the two filtering methods, green represents the Snapshot comparison while red represents filtering using API traces. It can be noticed the number of mutants extracted using Snapshot Comparison is equal to zero. As Mutexes represent objects and not "physical" files or database entries, they cannot be extracted by comparing two states of the Machine. Such task can be done by performing a memory dump of the system after the malware has finished executing. Due to the limited HDD space available for this project it has been decided to not enable the memory dump feature available with the Virtual Environment software in use.

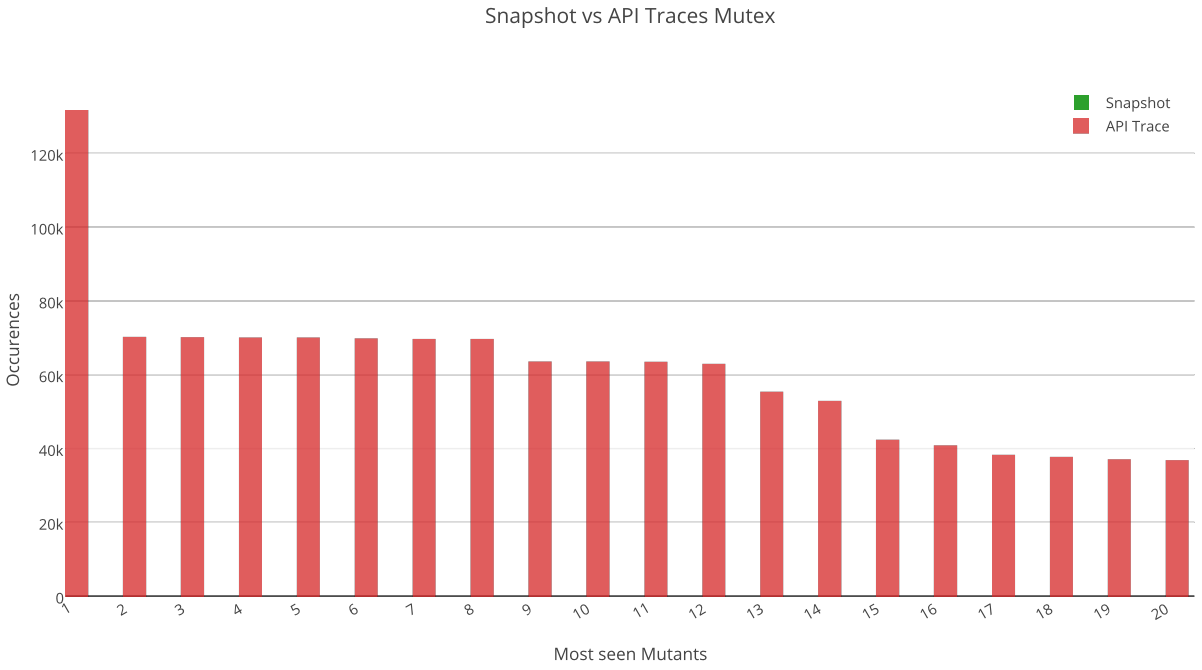


Figure 4.6: Barchart of 20 most seen mutexes using both methods.

4.6.3 Extracting Registry

Extracting Registry keys represents a more complex task than Files and Mutexes. Registry keys are usually divided in different sections, thus different arguments must be extracted from two different API calls to form the full path of a Registry key. The full list of API calls related to Registry Functions can be found on the Microsoft Developer page, see [Microsoft, 2008c]. The first step is to look for API calls that define the start location of the Registry, and it can have the following values:

1. RegOpenUserClassesRoot - Retrieves a handle for **HKEY_CLASSES_ROOT** key for a specified user on the machine.
2. RegOpenCurrentUser - Retrieves a handle for **HKEY_CURRENT_USER** key for the current user.
3. RegLoadKey - Retrieves a subkey from either **HKEY_USERS** or **HKEY_LOCAL_MACHINE**.

By combining the information extracted from the above API with the argument represented by *SubKey*, the full path of the registry key can be determined. In Figure 4.7, the x axis contains the registry key paths while the y axis contains the occurrences over all samples.

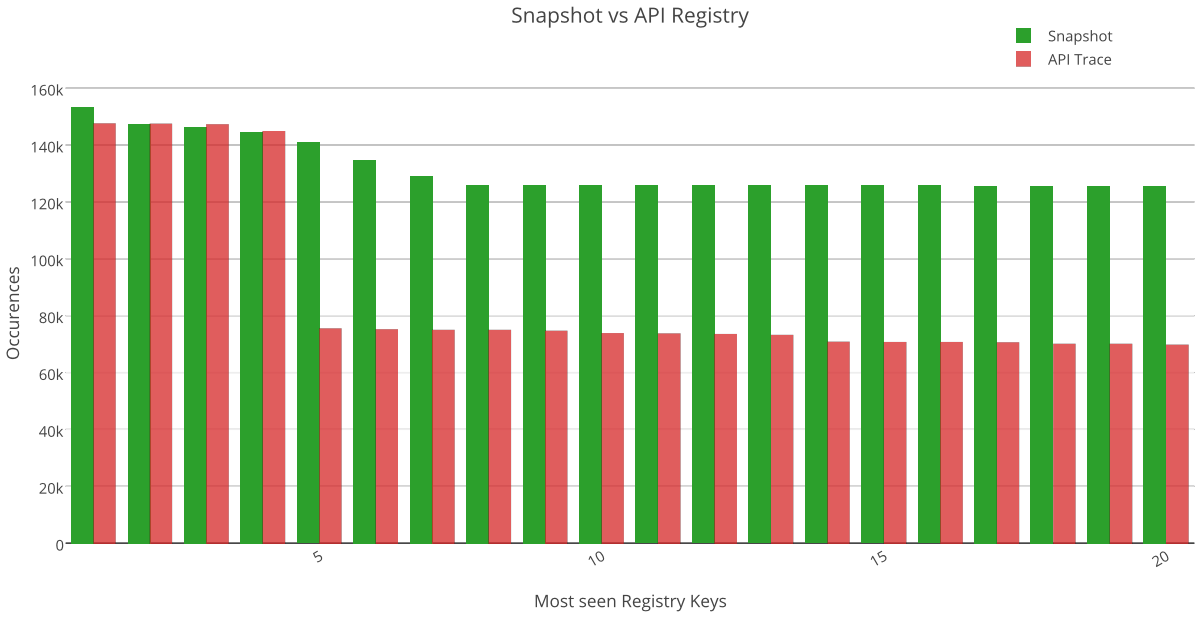


Figure 4.7: Barchart of 20 most seen registry keys using both methods.

Table 4.6 presents in the form of a table, the total number of distinct files, mutexes and Registry keys collected and filtered. It can be denoted that the number of distinct files is larger than using Snapshot Comparison. This can be explained by the fact that the API traces also include failed attempts. Overall the larger number of files suggests that collecting this feature using API traces will include information that was attempted but not recorded on the system. Mutexes, as described in previous sections, represents objects that control the sharing of access for processes. Compared with Files and Mutexes they do not represent physical objects that can be denoted from comparing two states of the Virtual Machine. The number of Registry keys has considerably been lowered by approximately 76%.

Properties	**Snapshot Comparison**	**API Traces**
Files	1,682,715	1,754,176
Mutexes	-	31,398
Registry	351,658	79,882

Table 4.6: Table of distinct features seen using both methods.

Due to the high amount of distinct values present in all of the collected features, it has been decided that only API calls will be used as features. The number of distinct API calls represents a finite number of possibilities and contain fair amount of information which also include the actions done with the presented features from Files, Mutexes and Registry Keys. Next section will describe the feature representation of

the API calls that will be used in unsupervised Machine Learning.

4.7 Feature Representation

This section covers the representations of the extracted features in Section 4.6 (p. 32) in order to be used for Unsupervised Learning. As the number of API representations described by successful, failed and their corresponding return codes, is relatively large, the data must be presented in a compact form without losing any potential discrimination between types.

The best solution, as used in previous project, would be to present the collected features in the form of a Binary or Frequency matrix, where information is denoted by numerical values. The next subsections will describe, in detail, how the matrices are created, and how they correlate with the collected behavioral data.

4.7.1 Binary Representation

The binary representation technique attempts to describe a certain feature by a simple 0 and 1 value. This provides limited but important information about each sample, giving an insight about its actions. Furthermore, as discussed in previous chapters, Successful and Failed API calls will be used, for each sample, to attempt a correct clustering of malware types. In order to describe the built Binary matrix representation for the API features, Equation 4.1 is presented as an example.

$$API_{bin} = \begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_m \end{array} \begin{array}{c} \begin{array}{cccc} API_1 & API_2 & \cdots & API_n \end{array} \\ \begin{bmatrix} 1 & 1 & \cdots & 0 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \cdots & 1 \end{bmatrix} \end{array} \quad [\cdot] \quad (4.1)$$

Figure 4.8: Binary Representation of API calls.

For each sample, a Binary representation is shown, describing if a certain API has been called or not during the analysis. The API calls are described on the horizontal axis and the number of samples are listed on the vertical axis. Rows with all values equal to 0 are not present as they represent samples that had no API calls. Such samples were not taken into consideration as it denotes that the injected program had performed no actions on the machine, as described in Section 4.3 (p. 20).

4.7.2 Frequency Representation

The frequency representation technique aims at providing a more detailed description about the chosen features. In comparison with the Binary representation it is also counting the number of occurrences of each API call. If a feature is not seen, then the number remains set to 0, otherwise it increments by one

ending up with the total number of calls. To illustrate the foundation of this representation, Equation 4.2 is presented as an example.

$$API_{freq} = \begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_m \end{array} \begin{array}{c} \begin{array}{cccc} API_1 & API_2 & \cdots & API_n \end{array} \\ \begin{bmatrix} 677 & 43 & \cdots & 0 \\ 4 & 8785 & \cdots & 51 \\ \vdots & \vdots & \ddots & \vdots \\ 414 & 27 & \cdots & 7397 \end{bmatrix} \end{array} \quad [\cdot] \quad (4.2)$$

Figure 4.9: Frequency Representation of API calls.

The values on the horizontal axis represent the number of occurrences for each API, while the values on the vertical axis represent each sample used. The Frequency matrix will be used, in this project, for representing the number of occurrences of Successful and Failed API calls, along with the corresponding error codes.

4.7.3 Sequence Representation

The sequence representation will take into account the order in which the APIs are called. Compared to previous project, current representation will include the whole spectrum of seen APIs by only allowing distinct values to be part of the representation. To improve performance of algorithms running this representation, the API values have been converted to numeric values. For example API *OpenKeyEx* is present in the full list of APIs at a certain index. The value of the index will be used to represent the API instead of the actual value. Equation 4.3 illustrates the contents of the representation described above:

$$API_{freq} = \begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_m \end{array} \begin{array}{c} \begin{array}{cccc} SEQ_1 & SEQ_2 & \cdots & SEQ_{157} \end{array} \\ \begin{bmatrix} 22 & 52 & \cdots & 21 \\ 24 & 84 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 3 & \cdots & 122 \end{bmatrix} \end{array} \quad [\cdot] \quad (4.3)$$

Figure 4.10: Sequence Representation of API calls.

As previous representations, the vertical axis represents the samples while the horizontal axis represented, in this case, the sequence number limited to the total count of distinct APIs seen in the current database. If samples do not utilize all 157 API calls then the remaining sequence columns that have not been filled will contain a value of -1 . This value has been chosen such that it will not interfere with any programming languages that will be used in future sections.

However, as it is believed that the Sequence of events performed by the malicious software depends mainly on the implementation of the malware and it can better describe families rather than types. This

representation will not be included if the evaluation of the clustering algorithm will be done on types.

4.7.4 Combining Representations

In order to use all of the presented feature representations, a combined representation is introduced where all of previous features are concatenated to a single matrix. This is simply done by merging the values of each representation to the corresponding samples. Equation 4.4 describes how the Combination matrix is represented.

$$Combination = \begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_m \end{array} \begin{array}{c} \begin{array}{cccccccccc} Passed_1 & \cdots & Passed_{157} & Failed_1 & \cdots & Failed_{157} & RC_1 & \cdots & RC_{137} \end{array} \\ \left[\begin{array}{cccccccccc} 22 & \cdots & 3 & 21 & \cdots & 2664 & 35 & \cdots & 533 \\ 52 & \cdots & 21 & 224 & \cdots & 123 & 45 & \cdots & 346 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 52 & \cdots & 21 & 224 & \cdots & 123 & 45 & \cdots & 346 \end{array}\right] \end{array} \quad [\cdot] \quad (4.4)$$

Figure 4.11: Frequency combination of API calls and Return Codes.

In order to better understand the contents of the combined matrix seen in Equation 4.4, it is split in three parts where each row represents the representation of a single malware sample.

1. Columns 1 to 157 represent the Frequency of the 157 Passed API calls.
2. Columns 158 to 315 represent the Frequency of the 157 Failed API calls.
3. Columns 316 to 453 represent the Frequency of the 137 Return Codes.

As the number of features in the combination representation is fairly large, feature selection needs to applied to reduce the number of dimensions. This is done by removing the non-relevant features or combining the relevant ones using various methods which will be presented in the following section.

INTENTIONALLY LEFT BLANK

Chapter 5

Clustering & Evaluation

This chapter will include decisions made with regard to Machine Learning algorithms in order to chose the best suitable methods that should correctly cluster malware behavior and evaluate its results. The chapter has been split in four main sections:

- Feature Selection in Section 5.1.
- Number of Clusters in Section 5.2 (p. 45).
- Unsupervised Machine Learning in Section 5.4 (p. 51).
- Evaluation Techniques in Section 5.5 (p. 56).

5.1 Feature Selection

This section will provide information about selecting the correct features with respect to their statistical and mathematical importance. The selection will be made from a list of algorithms available in WEKA, by assessing them in terms of performance and capabilities given the data set used for this project.

Feature selection represents the extraction process of a subset from the original feature list, that can better discriminate the data. This implies that the irrelevant and redundant features will be prone to removal as they hold no relevant information that can improve the models predictive performance. The benefits of using a Feature Selection algorithm before constructing the predictive model are, [Wikipedia, 2015b]:

- improved clustering/training duration.
- improved discrimination.
- increased understanding of data.

Feature selection algorithms are combining techniques of searching for the appropriate feature subset with evaluation techniques that have the goal of assigning scores or rankings to the selection as illustrated in Figure 5.1, see [Guyon and Elisseeff, 2003].

Selection of features in Unsupervised Learning represents an important problem due to absence of labels as pointed out in [Roth and Lange, 2003]. In order to more precisely select a subset of features, the use of labels, extracted by the Majority Vote between AV vendors in Section 4.4.1 (p. 25) and Microsoft labels, will be able to provide the best solution for better discriminating between malware types or families and thus will be used as comparison. Nevertheless, both supervised and unsupervised feature selection methods will be taken into consideration in the following subsections, after which the best suited algorithm will be chosen.

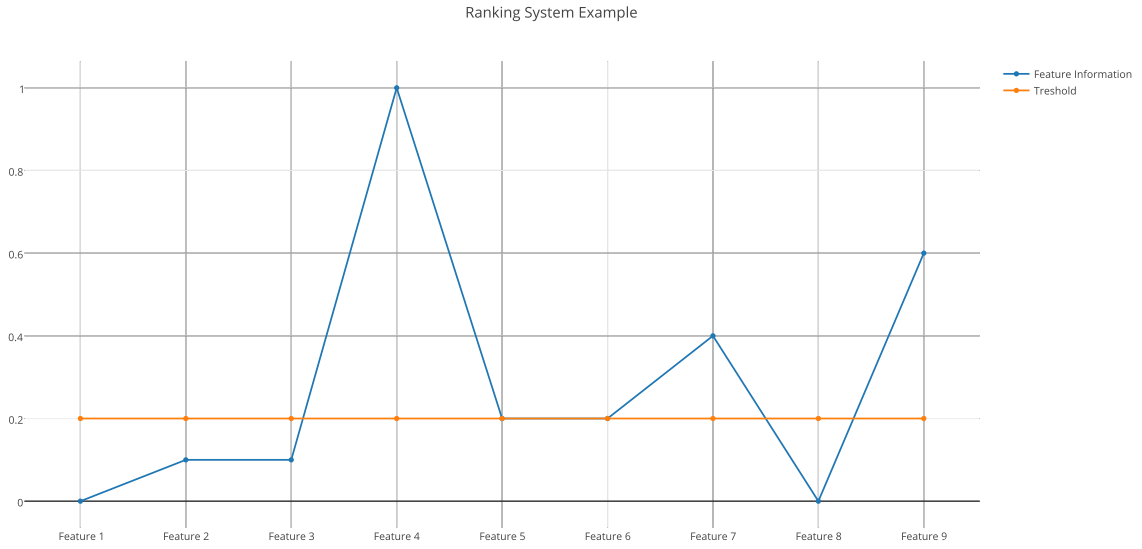


Figure 5.1: Ranking example for selecting features based on threshold(yellow line).

5.1.1 Principal Component Analysis

Principal Component Analysis (PCA) has the main purpose of reducing the dimensionality of the data by combining features and creating new features called principal components based on variance or co-variance, see [Abdi and Williams, 2010]. As it keeps the subset of features that are accountable for the highest variance it can also be used as feature selection. This method is closely related to the one used in the previous project where features with the highest normalized data were kept while the other ones were discarded. PCA simply uses linear algebra to compute and determine the components with the highest variance, see [Jolliffe, 2002]. The steps below are followed:

1. The whole d-dimension data-set is taken. If classes exist, they are ignored.
2. Compute the mean of each column representing the features.
3. Compute the co-variance matrix of the data set.
4. Determine the eighenvectors and corresponding eigenvalues using the means and co-variance matrix.
5. Decreasingly sort the eighenvectors by their eigenvalues.
6. Select the n largest eighenvectors representing the principal components.

An evaluation method can be used as a last step to provide scores to the sorted features. As the score will most probably be a value between 0 and 1, the selection can be made using a defined threshold. To summarize, PCA is able to find relevant structures from the original feature set, and discard features that don't have a great impact on a future decision.

5.1.2 Information Gain Selection

The Information Gain Selection evaluates and selects the appropriate features by measuring their information gain with respect to the class of each sample. Mostly used for building decision trees, it investigates

possible solutions that reach a certain state as fast as possible. These states are denoted as classes thus the use of labels is required in order for this method to work.

The Information Gain is closely related to Entropy. Entropy measure reflects the uncertainty within a data set X using the following formula, see [Kent, 1983]:

$$H(X) = -\sum_{s \in S} p(s) * \log_2 (p(s)) \tag{5.1}$$

where

- X represents the data set.
- S represents the classes.
- *p(s)*represents the marginal probability density function for the random variable S.

A value of 0 represent no uncertainty in the data which is relevant in the case of using only one class. Information gain can then be calculated by subtracting the uncertainty on a split done for a certain feature from the uncertainty seen over all samples and features. The following equation describes how information gain is calculated for each feature:

$$IG(A,S) = H(S) - \sum_{t \in T} H(S|t) \tag{5.2}$$

where,

- *IG(A,S)* represents the information gain by using feature A.
- *H(S given t)* represents the uncertainty or entropy for feature t using class S.
- *H(S)* represents the uncertainty for the class S.

After calculating the information gain for all features, a ranking system can then be generated based on the values in order to describe the most relevant features. Values close to 1 refer to features which always have an impact when classifying the data while a value of 0 represents no impact. Based on the information provided it can be concluded that classes have a great role in selecting the best features which makes the Information Gain unusable for unsupervised learning and clustering where classes are not available.

In order to compare the presented methods, Table 5.1 and Table 5.2 present the results in terms of number of features selected. Informational Gain has been tested on all three representations with majority vote classes, Microsoft classes and top five Microsoft classes. Furthermore, a 10-fold has also been applied to retrieve a more accurate feature selection. PCA on the other hand, has been tested on all representations with respect to variance and co-variance. All features have been selected using the Ranker algorithm with a threshold of zero representing no information gain. The sample set or classes column defines the subset of samples on which the feature selection has been applied and it is defined as follows:

- Majority classes refers to the labels generated via majority vote from multiple AV vendors as described in Section 4.4.1 (p. 25).

- Microsoft classes refers to the labels extracted from Microsoft as described in Section 4.4.1 (p. 28).
- Microsoft 5 types refers to the five most seen types in the sample set according to Microsoft AV. This represents a subset of the full samples only containing the selected five classes.
- Full samples set refers to complete analysis sample set. Also can be described as the super set of the analysis.
- 5 types, coincides with the Microsoft 5 types representing the subset containing the top five detected labels by Microsoft AV. The main difference between these sets is denoted by the absence of classes, being only used in PCA.

Method	Feature Set	Classes	Starting Features	Resulting Features
Information Gain	Failed API	Majority	157	104
		Microsoft	157	99
		Microsoft 5 types	157	104
	Passed API	Majority	157	140
		Microsoft	157	138
		Microsoft 5 types	157	143
	Return Codes	Majority	137	67
		Microsoft	137	67
		Microsoft 5 types	137	84

Table 5.1: Information Gain : Feature Selection comparison

Method	Feature Set	Sample Set	Starting Features	Resulting Features
Variance PCA	Failed API	Full	157	117
		5 Types	157	115
	Passed API	Full	157	115
		5 Types	157	116
	Return Codes	Full	137	115
		5 Types	137	114
Co-variance PCA	Failed API	Full	157	12
		5 Types	157	12
	Passed API	Full	157	18
		5 Types	157	17
	Return Codes	Full	137	3
		5 Types	137	3

Table 5.2: Principal Component Analysis : Feature Selection comparison

A significant increase in the number of features selected can be denoted in the variance PCA method in comparison with the Information Gain. This denotes that a large number of features, have values that vary over each sample making it possible to discriminate between groups. As described in Section 4.7 (p. 37), the selected features from all three Feature Representations will be combined to form a

Combination matrix that will be used in both finding the number of clusters and unsupervised learning. A decision has been made to make use of the co-variance PCA selected features as it represents the lowest number of features that can describe the data-set. It is assumed that the information within is enough to discriminate between types as they will contain both variance and co-variance information with respect to other samples or features from the same sample. It has to be mentioned that the features are now represented by the Principal Components that do not hold the frequency but instead they hold relationships between the frequency of features.

This section has described supervised and unsupervised methods of selecting features, where a decision has been made to use co-variance PCA. The following section will provide multiple approaches on how to select the number of clusters given the Combination representation denoted in this section.

5.2 Number of Clusters

This section will presented different methods of finding the appropriate number of clusters to be used in an unsupervised Machine Learning algorithm. The resulting number of features can be used in algorithms that require such an input or as an evaluation metric for algorithms that automatically find the number of clusters.

The decision of the number of clusters to be used on a data-set is the most frequent problem in clustering. This problem is also referred as finding k, corresponding to algorithms like k-means. The simplest approach for applying unsupervised machine learning is to use algorithms that do not require a set number of clusters. Such examples are Density-Based Spatial Clustering of Applications with Noise (DBSCAN), Ordering points to identify the clustering structure (OPTICS) or Expectation-Maximization (EM) algorithms. However, this section aims at finding methods that can define the number of cluster before applying ML on the selected features in order to evaluate the selection made on the malware data-set.

There are multiple methods that try to find the best amount of clusters required by the data-set, however only two have been chosen for review as they provide good performance for large number of samples. Other methods, like the Silhouette measure, have been also tested but due to the large data-set in use for this project they have not been used.

1. Elbow Method.
2. Gap Statistics.

Following subsections will describe the cluster selection methods and evaluate them based on ease of implementation in order to be used in a clustering algorithm.

5.2.1 Elbow Method

The Elbow method represents one of the simplest ways of trying to achieve the best number of clusters by examining the variance as a function of the number of clusters. As many other methods, it requires the use of a clustering algorithm, like k-means, to perform clustering and return the amount of variance each

cluster produces. It is expected that after the correct number of clusters reaches the marginal gain, the percentage will decrease or will increase at a much smaller rate than with less clusters [Thorndike, 1953].

In order to calculate the variance, Euclidean Distance is used to determine the distance between two points. The resulting calculation will represent a 2-dimensional vector containing information about inter and intra cluster similarities. In other words, will contain information about a point and the similarities with points inside the same cluster along with similarities with points belonging to different clusters. In Equation 5.3 the formula for calculating the intra cluster similarity for all points can be depicted.

$$D_k = \sum_{\mathrm{x}_i \in C_k} \sum_{\mathrm{x}_j \in C_k} ||\mathrm{x}_i - \mathrm{x}_j||^2 = \sum_{\mathrm{x}_i \in C_k} \sum_{\mathrm{x}_j \in C_k} ((\mathrm{x}_j - \mu) - (\mathrm{x}_i - \mu))^2 = 2n \sum_{\mathrm{x}_i \in C_k} (\mathrm{x}_i - \mu)^2 \tag{5.3}$$

where,

- C_k represents the cluster k.
- x represents some point in cluster C_k.
- D_k intra cluster sum of cluster k.
- n denotes the number of samples in cluster k.

Equation 5.3 has determined that the intra cluster difference between points is denoted by the squared difference between each point of the cluster and the mean. The final D, representing the distance measure will contain a matrix of the summed distances between the points and the center of cluster. However as the clusters may contain different number of samples, the distance matrix must be normalized. This will reflect a fair calculation for each sample and it is done using Equation 5.4.

$$E_K = \sum_{k=1}^{K} \frac{1}{2n} D_k \tag{5.4}$$

where,

- E_K - Explained Variance using K number of clusters.
- D_k - Euclidian Distance matrix for cluster k.

Explained variance variable E_K now contains a single value which denotes the metric of evaluation using K number of clusters. The Elbow method should iterate through a range of clusters in order to determine the optimal solution. It is expected that the explained variance should drop at a fast pace until it reaches the optimal solution, after which the values should decrease slower than before. To exemplify, Figure 5.2 (p. 47) presents the elbow method applied on the current data set. It can be seen that the results are ambiguous and no information can be easily extracted without setting some threshold of the decreasing slope which requires prior information about the optimal solution. Thus, other methods need to be researched that can automate the process of selecting k, representing the number of clusters.

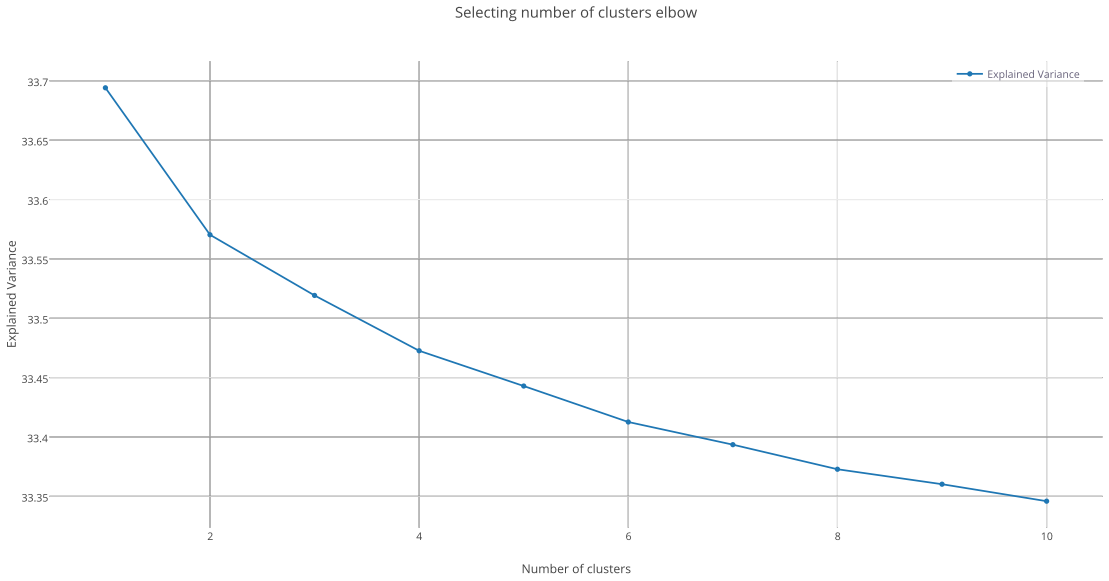


Figure 5.2: Explained Variance for multiple number of clusters.

5.2.2 Gap Statistics

Gap statistics represents a continuation of the Elbow Method, presented in [Tibshirani et al., 2001]. The idea behind this approach is to find a standardized comparison method between the log-likelihood of the data and a null reference. The null reference can be described as a random distribution with no obvious number of cluster, random. The estimation of the number of clusters is represented by the lowest $logE_K$ in comparison with the null reference curve. The Gap statistic is calculated using Equation 5.5.

$$\text{Gap}_n(k) = E_n^*\{\log E_k\} - \log E_k \tag{5.5}$$

where,

- E_K represents the Explained Variance for K clusters, also seen in the Elbow Method, Equation 5.4.
- E_n represents the reference distribution.

From this equation the optimal number of clusters can be extracted and it is defined as being the smallest k where the gap statistic of the previous value is larger then the gap statistic of the current value, see Equation 5.6. Furthermore Figure 5.3 (p. 48) represents the results of the implemented method for the data set in use where it can be seen that the Gap Statistic provides an automated and unambiguous way of selecting the appropriate number of clusters.

$$K = argmin\forall_k Gap(k) \geq Gap(k+1) - s_{k+1} \tag{5.6}$$

where,

- K represents the selected number of clusters.
- k represents the current number of clusters for which gap statistic is calculated.
- s_k represents the simulation error of the reference distribution.

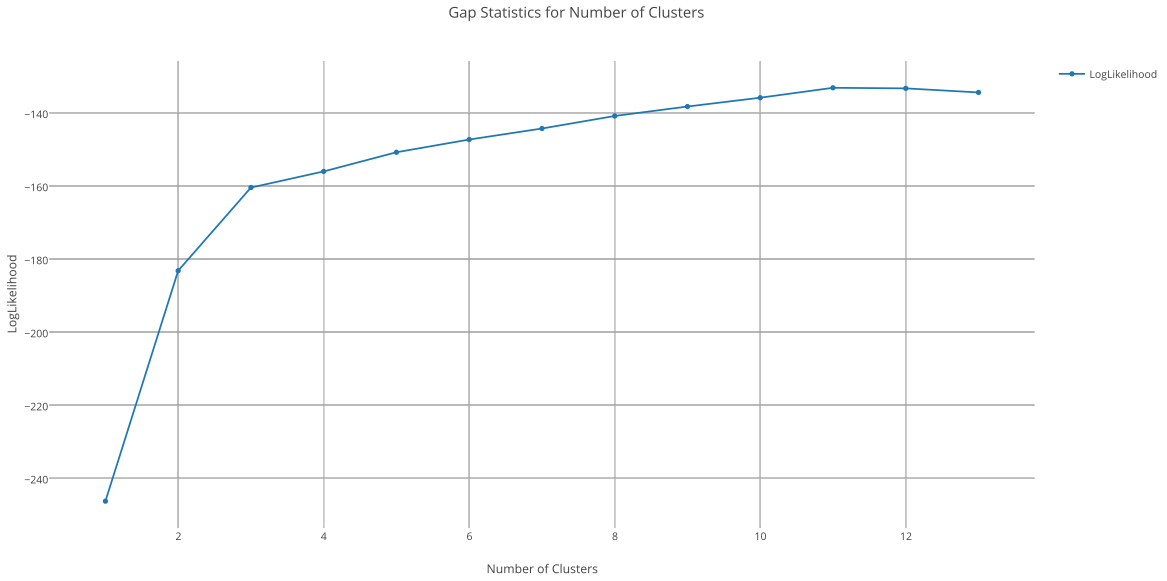


Figure 5.3: Gap Statistic for multiple number of clusters.

Comparing the two presented methods, the Gap Statistic has been chosen to define the number of clusters for future use in an unsupervised algorithm as it provides the project with an implementable way of solving this common problem. This section has presented a few methods of selecting k and in the following section, unsupervised algorithms will be analyzed and selected in order to choose the most appropriate method of clustering malware behavior.

5.3 Unsupervised Machine Learning

This section will describe the approach used in this project for applying unsupervised Machine Learning (ML) on the data. The goal of this approach is to find hidden structures or groups of data. It does so by using measures of similarity that are usually calculated using Euclidian distance.

There exist a large variety of clustering algorithms that aim at generating groups from the input data-set as accurately as possible. Different types of clustering algorithms can be distinguished by the methods used and can be classified as follows:

- Based on similarity or distance measure between input points. This method represents one of the traditional approaches and can be seen in k-means.
- Probabilistic where the sample set is assumed to be generated by some distribution. The Gaussian Mixture Model (GMM) is an example of such probabilistic method which is based on the EM algorithm.
- Hierarchical. This method can be further split into Agglomerative and Divisive.

Out of these, only three algorithms have been considered, k-means, EM and SOM, with respect to accuracy and performance under large data-sets that expand over a high amount of dimensions represented by the features, see [Abbas, 2007]. Their use expand not only in malware analysis but in many other

fields like: medicine, computer vision, etc.

In the following subsection, the three selected algorithms will be briefly explained after which a decision will be made. The selected algorithm will be used to cluster malware behavior and evaluate the results using the Majority Vote and Microsoft labels.

5.3.1 Clustering Algorithms

This subsection will now briefly explain the considered ML algorithms in this project after which one will be chosen and described in detail in Section 5.4 (p. 51).

k-means

The kmean algorithm is directly based on an article published by Hartigan, see [Hartigan and Wong, 1979]. The k-means clustering method aims at minimizing the average squared distance between points that are part of the same cluster. Given k, denoting the number of clusters, X denoting the data points and V denoting the set of centers, k-means algorithm does the following:

- Randomly select k cluster centers.
- Calculate the distance between each data point X_i and cluster center V.
- Assign the data point to the cluster center whose distance from the center is minimum of all other cluster centers.
- Re-calculate new cluster centers V.
- Re-calculate the distance between each data point and the newly obtained centers from step 4.
- If algorithm has converged stop, otherwise repeat from step 3.

The algorithm is easy to understand and performs clustering at a faster rate in comparison with other complex clustering algorithms. However, it does require to have a data set with well separated points to return an accurate grouping of the data. As there is no indication of such separation in the malware data, k-means is not guaranteed to be the best possible solution in clustering malware types or families. It also has to be mentioned that the results can be different with each run of the algorithm as the initial step of the kmeans algorithm depends on selecting a random location for each center. This problem can be resolved by controlling the randomness using a pseudo-random seed by selecting the centers from a known sequence of numbers. The kmeans implementation that uses seeds to select the centers is called kmeans++ and can lower the complexity of the algorithm, see [Arthur and Vassilvitskii, 2007].

Self-Organizing Map

The Self-Organizing Map (SOM) algorithm is based on the book published by Professor Teuvo Kohonen, see [Kohonen, 1989]. SOM was developed as a competitive learning algorithm, part of Neural Networks that contains a hidden layer. The algorithm has two phases: *Learning phase* and *Prediction phase*. The prediction phase works just like a classification algorithm where new data points are assigned to clusters generated in the Learning phase of the algorithm. The steps are:

1. Create the map using a predefined grid size of n by m. There will be $n * m$ number of nodes. Dimensionality of the data is non-linearly reduced to fit the grid size.
2. Initialize the weight of each node.
3. Choose a random input vector and present it to the map.
4. Find Best Matching Unit (BMU). The distance between the input vector and the weight of each node is calculated to determine the BMU.
5. Calculate the radius of the neighbors around the found BMU. With each iteration, the neighborhood size should decrease.
6. Modify the node weight such that its properties are closer to the BMU. The nodes further away from the BMU are slightly changed in comparison with the nodes closer to the BMU.
7. Exit if algorithm has converged, otherwise repeat from step 3.

EM Algorithm

The EM algorithm focuses on assigning a probability distribution to each sample, defining the clusters it belongs to. Just like DBSCAN, this method can select automatically the number of clusters it believes the data holds using a log-likelihood averaged over a 10-fold validation. These steps are as follows :

1. The starting number of clusters is manually set or will start from 1.
2. The data in randomly and evenly split into 10 folds.
3. Expectation and Maximization steps are applied on every created fold [Do and Batzoglou, 2008].
4. Log-likelihood is averaged over the 10 results.
5. Select number of clusters if the log-likelihood has stopped increasing otherwise increase number of clusters by 1 and go to step 2.

To conclude the listing of various unsupervised machine learning algorithms, the Self Organizing Map has been chosen to cluster the malicious behavior. The choice was made based on [Bação et al., 2005] which states that the input space is better explored by the Self Organizing Map. To consolidate his claims, he has compared kmeans with SOM on four different data-sets where SOM has outperformed kmeans in every aspect in terms of Structural Error, Class Error, Standard Deviation Error and Quadratic Error as Table 5.3 shows based on the IRIS data-set.

Algorithm	QuadraticErr	Std(Qerr)	ClassErr	StructErr
SOM	86.67	0.33	9.22	0
k-means	91.35	25.76	15.23	18

Table 5.3: Tests done on IRIS data-set by [Bação et al., 2005].

5.4 Chosen Algorithm

This section will go into more detail about the chosen algorithm. As decided in previous section, the algorithm that will be used to cluster the malware behavior data is **Self Organizing Map**. Each step of the algorithm will be depicted in detail, comparing the methods with other clustering algorithms. Furthermore a graphical illustration will be presented to better understand the shape and purpose of each created cluster.

5.4.1 SOM Overview

Self Organizing Map algorithm for multi-dimensional data relies on projections to a grid that keeps the topology of the original space. The idea behind this algorithm is closely related to kmeans where clusters are created based on some centroids. In this case the centroids are defined by the number of nodes which are inter-connected using a weight vector. Because of the weighting of the edges the problem can be seen as an elastic net that takes the shape of the data.

As SOM is trained iteratively, the vectors are chosen one by one from the input space. The distance between the input vector and each created node on the grid is calculated based on their weights. This becomes a minimization problem with the objective function depending on the Euclidean distance, from which the minimum is selected representing the BMU. Furthermore the weight of the nodes are updated after finding the appropriate BMU, and the nodes are moved closer to the projected vector. The further the nodes are from the BMU the lesser the weighted update becomes, or the lesser they move towards the BMU.

This iteration continues until the map converges and no updates are seen. The clustering ends by creating a radius around close nodes giving the total number of clusters for a particular data-set. It has to be mentioned that results differ from grid size to grid size, thus an appropriate grid size has to be used in order to reflect the maximum number of clusters. For example, if a grid size of 2 by 2 is set, this limits the SOM algorithm to a maximum of four clusters, as the number of nodes is also four. As the grid size increases, the number of clusters will converge as the nodes will move closer to their appropriate BMUs. However the usual results in terms of number of clusters is mainly defined by the grid size.

Figure 5.4 depicts a very high level design of the Self Organizing Map. Given the sample set, SOM projects it to a predefined grid of size n by m and then it forms clusters based on grouped nodes. Next subsections will explain in detail all the steps and properties of the Self Organizing Map and how are they contributing to solving the clustering problem at hand.

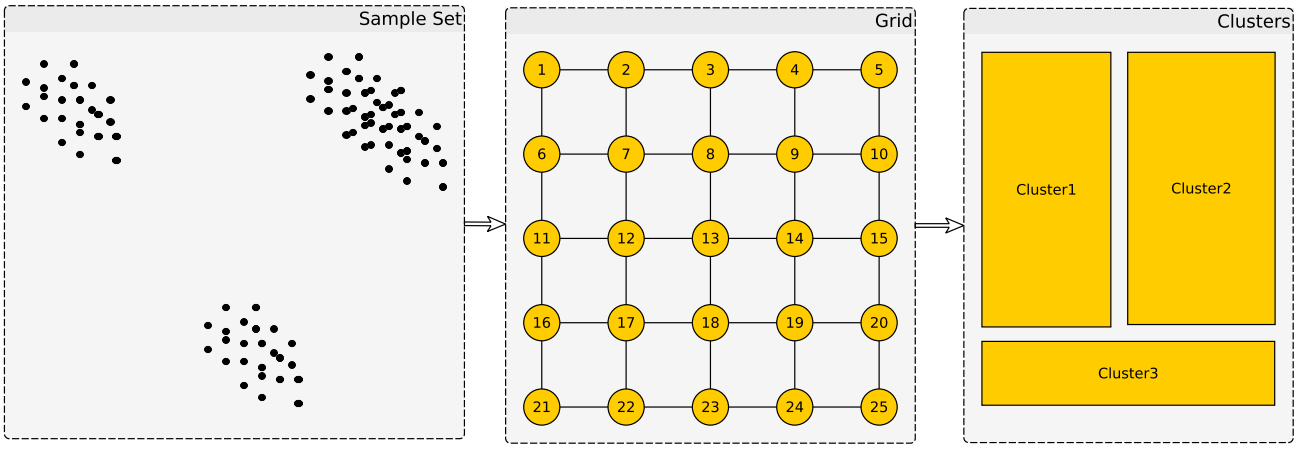


Figure 5.4: Steps describing the Self Organizing Map

5.4.2 Dimensionality Reduction

As stated before, SOM relies on reducing the dimensions of the data to one or two dimensions. First comparison can be made with Principal Component Analysis which represents the most commonly used method of reducing dimensionality in unsupervised learning. PCA aims at transforming the sample space into a lower-dimension space that is **linearly** uncorrelated. This method might be successful for data-sets where clusters can be distinguished linearly, however, the uncertainty and randomness in the analysis of malware behavior make this task close to impossible. Thus a more robust and accurate dimensionality reduction technique is required that can keep the properties of the input sample space and at the same time doing so in a non-linear way.

Even though SOM transforms the input space into a two-dimensional space, the weight of the nodes will still have the same dimensionality of each input vector. This is defined as

$$I = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ \vdots \\ X_i \end{bmatrix} \Longrightarrow W_j = \begin{bmatrix} W_{j1} \\ W_{j2} \\ \vdots \\ \vdots \\ W_{ji} \end{bmatrix} \tag{5.7}$$

where,

- I represents the Input Vector or a single sample.
- W represents the Weight Vector.
- i represents the number of dimensions of the the selected sample. This can be interpreted as the number of features.
- j represents the node corresponding to the SOM grid.

This methods provides a great advantage when dealing with malware analysis as no crucial information is lost during the task of dimesionality reduction. An example of SOMs method for dimensionality reduction

can be depicted from Figure 5.5, where input vectors x_1 and x_2 update the weights of all nodes based on their features.

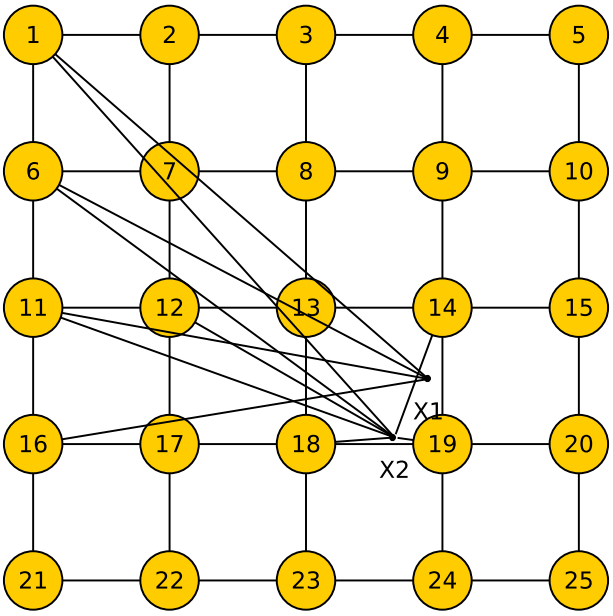


Figure 5.5: Dimensionality reduction example

5.4.3 Best Matching Unit

After each input vector has been projected to the SOM grid, the Best Matching Unit or BMU is determined representing the winning node or neuron. The BMU will directly affect the shape of the map, as neighboring nodes will move closer to it by updating their weight vectors.

The BMU is calculated using the Euclidean Distance between the input vector and the weight vector of each node. The BMU of each input vector represents the minimum distance to one of the nodes, thus it is described as a minimization problem. The Euclidean Distance, for this problem, is described as follows

$$E = \sqrt{\sum_{i=0}^{i=n}(X_i - W_i)^2} \tag{5.8}$$

where,

- X represents the current input vector.
- W represents one of the nodes.

It has to be mentioned that the above equation has to be done for all the nodes in the map. After the distances have been calculated, the BMU is define as the input vector that has the minimum distance to one of the nodes, as described in Equation 5.9.

$$BMU = argmin\forall_n E_n \tag{5.9}$$

where,

- E_n is the Euclidean distance of node n for the current input vector.

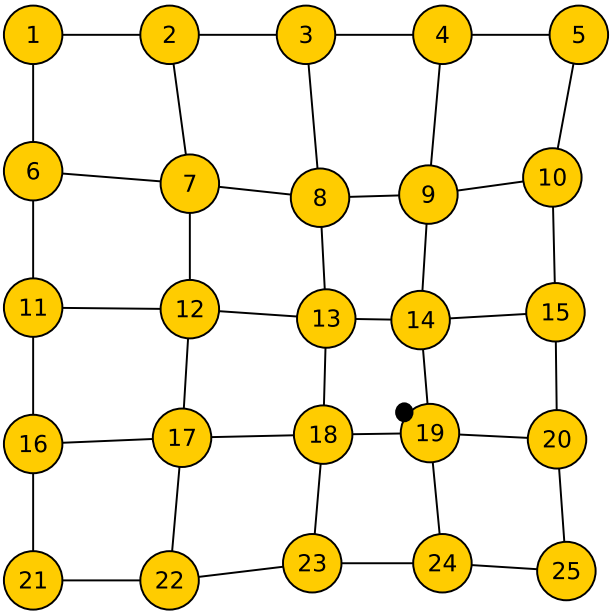


Figure 5.6: Graph form after weights have changed based on selected BMU

When selected, each BMU will have a radius, defining its neighbors, see Equation 5.10. The initial value of the radius will include almost all nodes of the map. As the algorithm progresses, this radius will decay denoting that the number of updated nodes will get smaller. This implies that the number of samples in use could define a partitioning of the map that would better describe the data-set. The decaying rate of the radius along with the modification of the nodes weights are controlled by a variable called *learning rate* which is described in the next subsection.

$$\sigma(t) = \sigma(0) \cdot e^{-\frac{t}{\lambda}} \tag{5.10}$$

where,

- $\sigma(t)$ represents the radius of the neighborhood at time t.
- $\sigma(0)$ represents the radius of the neighborhood at time 0. Initial condition covering almost all nodes of the map.
- t represents the current iteration.
- λ represents a time constant defined in Equation 5.12.

5.4.4 Learning Rate

The Learning rate as explained in [Stefanovič and Kurasova, 2011] has the purpose of minimizing the changes done to the weights of the nodes and radius of BMUs as iterations increase. This means that, as times goes on, BMUs have less effect on the nodes, giving the algorithm less time to converge, but at

the same time, define a more specific partition of the data as the radius shrinks and the neighbors of the BMUs will decrease.

The learning rate is a function of time and could be of different types:

- Linear.
- Inverse-of-time.
- Power series.
- Heuristics.

Each method introduces a different learning rate decay however, for simplicity, this project will only use the linear one. Thus the learning rate will decay linearly over time given a set number of iterations as shown in Equation 5.11.

$$L(t) = L(0) - e^{\frac{t}{\lambda}} \tag{5.11}$$

where

- $L(t)$ represent the learning rate at time t.
- $L(0)$ represents the learning rate at time 0. Also known as the initial condition.
- t represents the current time or iteration.
- λ represents a time constant defined by the properties of the map.

The time constant used in the calculation of the learning rate relies on the total number of iterations along with the radius of the map, see Equation 5.12.

$$\lambda = \frac{numberOfIterations}{mapRadius} \tag{5.12}$$

5.4.5 Clustering

After all iterations have finished and the algorithm has converged, clustering is possible based on the instances and their respective BMUs. It has to be mentioned that the grid size of the Self Organizing Map has an effect of the number of clusters. Given a small grid size the projection will be done in a more restrained space, not allowing the representation to grow. It can be discussed that a smaller grid size can give a more general representation of the data while a bigger grid size can capture more detail by creating a larger number of clusters.

Self Organizing Map, used alone. will generated the same number of clusters as the number of created nodes. If a larger grid size is used, that exceeds the expectation for the number of clusters, nodes created by SOM can be used as input to another clusterer that will group the nodes and not the input vectors as seen in Figure 5.7 (p. 56). However, this project will focus on using only SOM with a fair number of nodes that would not require a second clusterer for grouping.

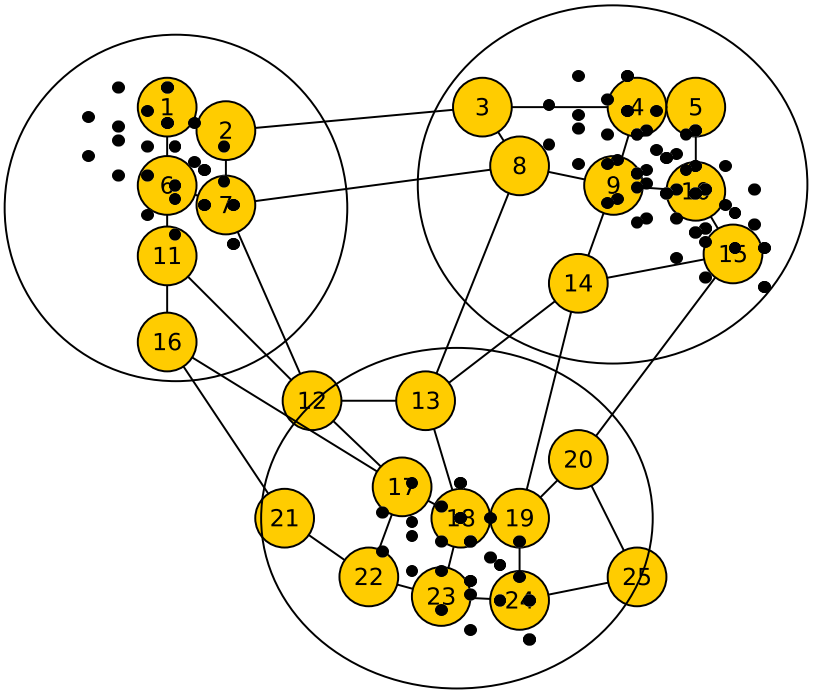


Figure 5.7: Steps describing the Self Organizing Map.

This section has covered the details of the unsupervised ML algorithm Self Organizing Map. The next section will present the methods of evaluation for the output of this algorithm.

5.5 Evaluation Techniques

In order to provide a measure of how good the unsupervised algorithm has clustered the provided instances, different methods are taken into consideration. In comparison with the methods used in last project, unsupervised learning provides a much smaller set of tools that can be used for evaluation. This section will describe how the clusters are labeled and how those labels are evaluated.

5.5.1 Cluster Labeling

Cluster labeling, in unsupervised learning, is done by a majority vote between the instances of a cluster. Table 5.4 illustrates how each cluster is labeled. It has to be mentioned that the correctness of the labels will greatly affect the metrics used to evaluate the classifier.

Cluster / Labels	C1 - Trojan	C2 - Backdoor	C3 - Adware	C4 - ?
Trojan	40000	3000	2000	200
Backdoor	20000	4000	3000	1000
Adware	2000	3000	50	0

Table 5.4: Assignment of Clusters labels

First of all, it can be denoted that clusters are labeled based on the number of instances seen with a particular label. In this example, as Trojan instances are more present in cluster one than any other types, the cluster is assigned the Trojan label. It can also be noticed that each label can only be assigned to one cluster. If there are more clusters than labels, then the instances falling in the unlabeled cluster will automatically be evaluated as incorrectly classified instances, as defined in Section 5.5.3. Cluster three on the other hand, represents the worst case scenario where, the cluster is named based on the class with the lowest number of labels as the other classes have already been used to name other clusters.

5.5.2 Evaluation Matrix

An Evaluation matrix is used to graphically describe the results of clustering after the created clusters have been labeled. In this project it will be used to analyze which labels had a chance at labeling a certain cluster as well as observing how the labels are spread across multiple clusters. It has to be mentioned that the Evaluation matrix cannot be used to calculate metrics like True Positive, False Positive, True Negative or False Negative as the correctness of the labels is not known and at the same time the number of cluster could be different from the number of labels. Comparing with supervised learning where, labels are assumed to be the truth of the training data, in unsupervised learning the data is defined to be the ground truth. One can assume that evaluation matrix can be used for evaluating the classes and not the clustering algorithm.

An example of an evaluation matrix is presented in Table 5.5, based on a subset of labels used in this project.

Cluster / Labels	Trojan	Backdoor	Cluster 3
Trojan	40000	3000	10000
Backdoor	20000	4000	200

Table 5.5: Evaluation matrix

The table described the labels of instances in the columns while, for the rows, it represents the assigned cluster of the instance. The naming of the clusters is done by majority vote as described in Section 5.5.1 (p. 56). This matrix will be used to analyze the contents of the clusters based on the provided labels from Microsoft or Majority Vote. It will provide an idea on the differences between the information stored in the behavioral data of the clusters and the labels retrieved from different AV vendors.

5.5.3 Incorrect Classification

Incorrect classification metric is presented as a percentage defined by incorrect labels inside clusters. This is calculated after the majority vote has taken place and it can be denoted in Equation 5.13 from the

evaluation matrix presented in Section 5.5.2 (p. 57).

$$Incorrect = IncorrectClassified/TotalSamples \tag{5.13}$$

where,

- IncorrectClassified represents the number of instances that are not in a cluster with the same label.
- TotalSamples represents the total number of instances in the data-set.

5.5.4 K correctness

This evaluation metric will be used to determine if the number of clusters selected with the method chosen in Section 5.2 (p. 45) is correct with respect to the AV labels used. Based on the information provided by AV labels, the number of malware types is known beforehand. This number must match with the number of selected cluster in order to determine if there is a link between the two methods.

Gap Statistics results will be evaluated based on the number of distinct type labels extracted from AV labels. The number of classes seen in both labeling techniques are seen in Table 5.6.

AV	Number of Types
Microsoft	5
Majority Vote	8

Table 5.6: Number of types.

This section has presented the evaluation methods for the chosen unsupervised algorithm. The section will present a summary of the choices made in this project.

5.6 Summary of Chapters 4 & 5

This section provides a short summary of the choices made in Chapter 4 and Chapter 5. By the end of this section the final system architecture will be presented and will be used in Part III for implementing the system.

Based on the sub-problems presented in Section 3.1 (p. 13) the following answers have been identified:

1. **Which features should be used?**
   In Section 4.6 (p. 32) it has been decided to use API calls along with their appropriate Return Codes. Furthermore API calls have been split into Passed and Failed API calls.

2. **Which feature representation should be used?**
   In Section 4.7 (p. 37) it has been decided that the Frequency representation should be used.

3. **How to perform majority vote on unstructured AV labels?**
   In Section 4.4 (p. 23) it has been decided that Levansthtein distance on AV labels should be used as majority vote to select the most appropriate label

4. **Which method of defining the number of clusters should be used?**
   In Section 5.2 (p. 45) it has been decided that the gap statistic will provide the information needed to select the appropriate number of clusters given a data-set.

5. **Which unsupervised ML algorithm should be used for clustering?**
   In Section 5.3 (p. 48) it has been decided that Self Organizing Map should be able to successfully cluster the collected malware behavior data.

Based on the decisions made in Chapter 4 and Chapter 5 the final system architecture has been created where Cuckoo Sandbox will analyzed the behavior of malware and store the information in MongoDB. After features are extracted, selected and represented in a Combination Matrix form they are passed to the Self Organizing Map algorithm to created appropriate clusters. The following Chapter represents the implementation of the system that is based on the final system architecture depicted in this chapter and presented in Figure 5.8.

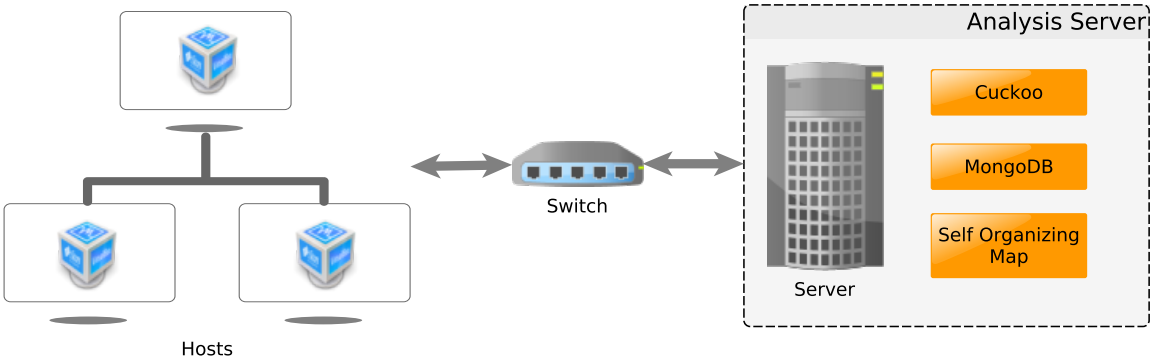


Figure 5.8: Final System architecture built on the decisions made in Chapter 4 and Chapter 5.

INTENTIONALLY LEFT BLANK

Part III

Design & Implementation

Chapter 6

System Design & Implementation

This chapter tackles the problems of implementing the design proposed for this project. A short description of the programming language will be introduced followed by detailed explanation of the implemented modules of the system.

6.1 Programming Language

This section will present the requirements in terms of Pre-Processing, Data Storage, API, Wrappers, Machine Learning and Documentation from which a programming language will be chosen to provide the coding environment for this project. The comparison is made between the most commonly used languages: Java and Python. To motivated the selection of the two, Python represents a straight-forward language that is easy to understand and community driven while Java represents a solid platform that is used to power most of today's hardware and software, including the ML toolbox used for this project, WEKA.

Following subsections will go in details about the requirements imposed by selections made in the Technical analysis part of the report, presenting the advantages and disadvantages of using one of the two Objected Oriented programming languages for implementation.

6.1.1 Pre-Processing and Data Storage

As presented in Section 4.3 (p. 20), the data collected from Cuckoo and the data storage of the selected DBMS follow the JSON structure. Thus, it is essential to choose a programming language that can easily manipulate this structure with low system requirements. As JSON represents a lightweight format it is supported by mostly all programming languages. Java makes use the *JSONParser* function of the external class *json-simple* to convert JSON formatted string representations to key-pair objects. Python on the other hand provides libraries that can manipulate JSON via its native dictionary data structure. It support nested structures, allowing multiple object types to be stored within. The main difference between the two languages for this requirement is that Java requires explicit type casting when retrieving information from the data structure while Python can automatically remember the type stored.

In order to process the data, it might be required to transfer information from one module to another. As the system is intended to run in a flow, or separately, it is essential for the chosen programming language to be able to easily transfer data between modules. Python provides such a capability by using a library called *pickle*, which represents a fast and efficient way of serializing and de-serializing objects. This refers to saving the object to an external file, converting in to a stream of bytes. Java, C# and many other languages also have support for serialization however they do not provide a plug-and-play solution

like the *pickle* library in Python. The same difference can be noticed here between Java and Python when de-serializing data where Java requires explicit casting of the object type stored while Python will automatically remember the stored type.

Given that most of data is represented as strings, it is important for the programming language to be able to support fast string manipulation. The majority of the Object-Oriented programming languages like Java, Python, Perl etc, provide easy to use string manipulation functions that can perform complex tasks. Python for example converts any series of characters to Unicode, which represents a consistent encoding and it is defined as a computing industry standard. Same functionality can be achieved using Java as it also provides a large number of native functions that ca trim, split, convert to upper or lower case and extracted sub-strings.

6.1.2 API and Wrappers

Support for executing external programs is a requirement for the program language that will be selected, granting the possibility to execute, from each module, queries to MongoDB and WEKA toolbox.

Python has the possibility of executing external programs through its native library *subprocess*. It creates a child process with which it can interact during its execution, providing the possibility of running it in parallel. This allows the modules that will be implemented to control WEKA via command-line and interact with it if needed. MongoDB provides a library called *pymongo* which connects to the DBMS allowing it to query the database directly from the module. The output of the query is directly translated to native Python data structure making the access to the output an easy task.

Java also has the possibility of executing external programs using the native class *Runtime*, however as WEKA is written using this programming language it can execute the functions without creating a process. This provides a great advantage over Python as the functions of WEKA could be directly imported in the modules. MongoDB provides a Java Driver for connecting and executing queries but it does not represent a plug-and-play solution in comparison with Python.

6.1.3 ML libraries

The need of ML libraries within the selected programming language provides a great advantage in terms of run-time and complexity. Being able to utilize Machine Learning algorithms, feature selection algorithms, label encoding and data visualization techniques using native data structures of the selected programming language removes the need of utilizing different files and storage alternatives as done in previous project. Java, as mentioned before, does not need such external libraries as it can directly make use of the functions provided by WEKA. On the other hand, Python Community provides a large number of libraries that can perform the aforementioned actions. The Machine Learning Python library *scikit-learn* [Scikit, 2015], combines the utility from three other libraries, and it is able to efficiently perform classification, clustering, visualization and dimensionality reduction:

- SciPy - library for mathematics, science, and engineering along with some functions for ML clustering found in *scipy.cluster*. See [Jones et al., 01] for more information about the library.
- NumPy - library containing a vast number of tools for manipulating large multi-dimensional arrays, linear algebra and more, [NumPy, 2015].
- MatplotLib - the library provides an easy way of visualizing data, making it able, if possible, to determine patterns. It also provides functions to create confusion matrices, hit maps and cluster maps required for the Self Organizing Map algorithm.

6.1.4 Documentation

Both programming languages provide solid and well-defined documentations for each class or library in use. The documentation provides detailed information of how the functions can be used, what the functions are achieving and what are the required parameters along with a structure of their outputs.

Requirements	**Java**	**Python**
Pre-Processing	✓	✓
Data Storage	✗	✓
API and Wrappers	-	✓
ML libraries	✓	✓
Documentation	✓	✓

Table 6.1: Filtering methods comparison

Summing up all the capabilities of the compared programming languages in this section, Python has been chosen to implement the modules required for this project mostly due to its native data structures that make the accessing and storing of data an easy task. This section has provided information about the programming language used for this project, next section will include an overview of the implemented modules along with an overall flowchart of the system.

6.2 System Design

This section will introduce the modules designed and implemented for this project. Each module will be presented in terms of required input and expected output. By the end of this section, the overall flowchart will be presented which defines the flow of the system.

6.2.1 Requirements

Malware Analysis module defines the first module of the system where the malware samples are injected to clean Virtual Environments in order to be analyzed. The module should be able to correctly control the analysis queue, pre-process the JSON output and populate the database following the specifications presented in Section 4.3 (p. 20). As the system is not designed for detection, the input samples need to be malicious.

Labeling module is responsible for correctly extracting labels from various AV vendors as well as for computing the Majority Vote labels from all AVs using tokenized Levensthein distance as presented in Section 4.4.1 (p. 25). Its expected output is to correctly update the database with the extracted labels for easier access during Feature Extraction.

Feature Extraction module is responsible for correctly extracting features from the data provided by Cuckoo and stored in MongoDB. The input of the module needs to follow the specifications of the Data Structure presented in Section 4.3 (p. 20). Based on the decisions made in the Technical Analysis part of the report, its expected output is represented by three matrices containing the frequency of Passed and Failed APIs along with their appropriate Return Code frequencies as presented in Section 4.7 (p. 37).

Feature selection module is responsible for correctly selecting the most appropriate features that can better characterize the behavior of the malware based on variance and co-variance as described in Section 5.1 (p. 41). Its input is represented by the output of the Feature Extraction module and the expected output is represented by a Combination matrix that contains the best suited features following the specifications presented in Section 4.7 (p. 37).

Number of clusters module is responsible for finding the optimal number of clusters given the data-set from the output of the Feature Selection module following the specifications presented in Section 5.2 (p. 45). Its output is represented by a single number, defined as the optimal number of clusters, that can then be applied in the Machine Learning module or used for evaluation purposes.

Machine Learning module is responsible for correctly applying Self Organizing Map on the data-set provided by the Feature Selection Module as presented in Section 5.4 (p. 51). The output, consisting of the created clusters in ARFF format, will then be merged with the output of the Number of clusters module in order to be evaluated in the Results part of the report.

In this subsection the modules that will be implemented have been presented in terms of required input and expected output. The flowchart of the system can be depicted in Figure 6.1. It has to be mentioned that the system has been designed to work in a flow and different modules cannot be run stand-alone unless previous modules have been executed beforehand. It can be denoted that the output of the Number of Clusters module is split as it has more than one purpose:

- As input to ML algorithms that require to manually set the number of clusters.
- For evaluation of ML algorithms that automatically determine the number of clusters.

6.2.2 Overall Flowchart

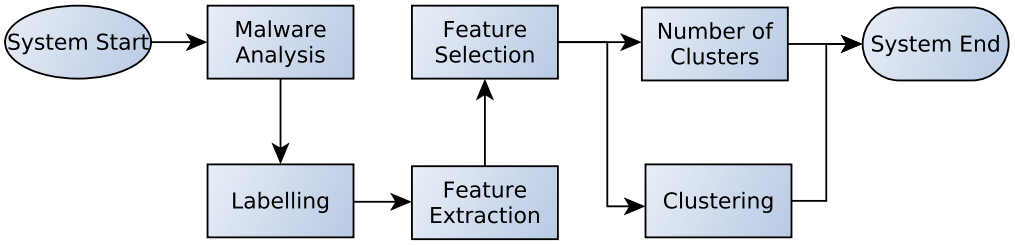


Figure 6.1: Overall flow chart of the system.

This section presented the requirements of the system along with an overall flowchart describing the order in which the modules are called and executed. Next sections will go into the details of each module, defining the main idea of the design along with the most important parts of the implementation code.

6.3 Malware Analysis Module

This section will describe the design and implementation of the Malware Analysis Module. This module is accountable for controlling VMs, injecting malware samples for analysis, populating and manipulating the MongoDB DBMS with the behavioral data. It relies mainly on the modified version of Cuckoo Sandbox, inserting its results in MongoDB. The following flow chart, presented in Figure 6.2 describes the flow of events for this module.

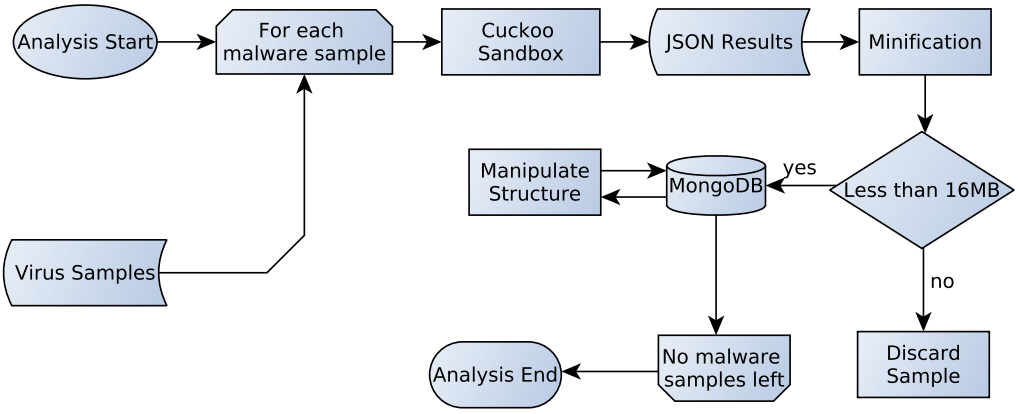


Figure 6.2: Flowchart for Malware Analysis Module

Cuckoo Sandbox represents the open-source program used for analyzing software in a fast and efficient way. Each malware sample is read from the disk and added to Cuckoo queue, which then assigns it to a free, uninfected Virtual Machine for analysis. Due to the high number of VMs in use, the processing of results and creation of the JSON files are not done in parallel, instead Cuckoo has been configured to wait until all samples have been injected. This has been done by setting a value of false to the *process_result* variable in Cuckoos configuration file as seen in Code snippet 6.1.

Code snippet 6.1: Cuckoo configuration

```
# Enable processing of results within the main cuckoo process.
# This is the default behavior but can be switched off for setups that
# require high stability and process the results in a separate task.
process_results = off
```

After the results of the analysis have been collected, Cuckoos processing module will incrementally process the results and create one JSON for each and every sample. Due to software limitations imposed by the selected DBMS, all imported JSON files must have a size smaller or equal to 16MB. In order to fulfill the requirements for more samples, each JSON file has been modified by removing unnecessary characters like new line, spaces or tab characters. This process is done by the minification sub-module which makes use of the JavaScript node called *json-minify*. Example of minification can be see in Code snippet 6.2. As the number of samples has greatly increased from the previous project, the process has been threaded to make use of all possible cores provided by the server.

Code snippet 6.2: Minification thread example

```
#!/bin/bash
# Query SQL queue
list=$(mysql -u root -p123 -e "SELECT id from malware.tasks WHERE status='reported' LIMIT
    $limit OFFSET $offset")
# for each task in the result
for split in $list
do
   reportID=$split
   # remove redundant information. ~ 50% reduction in size
   json-minify path/to/malware/json > /path/to/malware/minified 2>&1
done
```

After all the JSON files have been minified, they are ready to be imported to MongoDB for analysis. Before import, all files are verified in order to meet the 16MB requirement. If they exceed this software limit, the samples are discarded to prevent any errors during the import process. The samples are imported to the collection *analysis* of the database *cuckoofull* using the bash script seen in Code snippet 6.3

Code snippet 6.3: MongoDB import of minified JSON files

```
#!/bin/bash
# Query SQL queue
list=$(mysql -u root -p123 -e "SELECT id from malware.tasks WHERE status='reported'")
for split in $list
do
   # import JSON to DB cuckoofull and collection analysis
   mongoimport --host 172.26.12.229:27017 -d cuckoofull -c analysis --file /path/to/jsons/
       $split.json
done
```

MongoDB Before starting behavior analysis of the samples, pre-processing of the collected data will be done to remove any unnecessary information, as described in Section 4.3.2 (p. 21). In order to do so, a new collection has been created following the data structure presented in Figure 4.3 (p. 22). The pre-processing consists of the following actions:

1. Collect only relevant information from the behavior of the malware.
2. Remove samples with zero processes created.
3. Remove samples with zero API calls.

In order to extract only the relevant information, an aggregation command has been issued in MongoDB to project the relevant data matching the samples that do not have an empty process or API list. The command can be seen in Code snippet 6.4.

Code snippet 6.4: MongoDB aggregation

```
db.analysis.aggregate([
  {$project : {"virus" :  "$target.file.name", // project only needed information
         "info"  :  "$behavior.processes.process_name",
         "api"   :  "$behavior.processes.calls.api",
         "category" : "$behavior.processes.calls.category",
         "status" : "$behavior.processes.calls.status" ,
         "return_code" : "$behavior.processes.calls.return" ,
         "name" : "$behavior.processes.calls.arguments.name" ,
         "value" : "$behavior.processes.calls.arguments.value"}
         },
         {$match : {"info" : {$ne : [ ]}}}, // remove samples with zero processes
         {$match : {"api" : {$ne : [ ]}}}, // remove samples with zero API calls
         {$out : "APIradu"}], // output to new collection
         {allowDiskUse : true}) // allow use of HDD for memory extensive queries
```

The modified version of Cuckoo along with MongoDB configuration, minification and analysis bash scripts can be found on the CD in `..\Code\MalwareAnalysisModule\` path.

This section has presented how the malicious software is analyzed and imported to the database, preparing the data for the next module representing the Labeling Module.

6.4 Labeling Module

This module is responsible for correctly extracting relevant AV labels from the detection results provided by VirusTotal for each sample. As discussed in Section 4.4 (p. 23), there exists inconsistency in the AV labeling of malicious software and a temporary solution has been found by extracting the labels based on a Majority Vote from multiple AV vendors. This module is based on the flowchart presented in Figure 6.3.

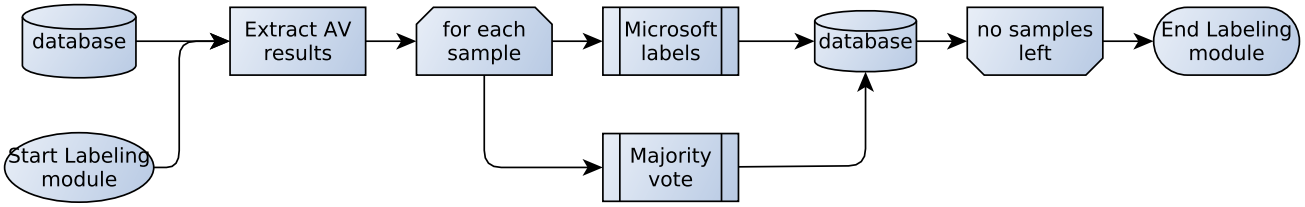


Figure 6.3: Flow chart for the Labeling Module

In order to analyze the labels provided by VirusTotal, the scan results for all AV vendors have been extracted and saved to a pickle serialized object. This has started by creating a list of all unique AVs available in the analysis, see Code snippet 6.5.

Code snippet 6.5: Extract list of AV vendors from database

```
# query db and retrieve the first sample
# from analysis collection
query = db.analysis.find_one()
# Create a list of AV vendors
listOfAV = list(query['virustotal']['scans'].keys())
# Number of AV vendors
leng = len(listOfAV)
```

The created list of AV programs will be used as header information for defining the correct index of each label. The index is then used to insert the appropriate labels for each sample and prepare the data for the Majority Vote process as well as for Microsoft label extraction. Code for the process of saving the labels to a serialized item will not be shown in this section, however it can be found in Appendix A (p. 112). The resulting file is also made available on the CD in the `..\Data\` folder.

After the *pickle* file has been created, the module iterates through all samples and performs the following tasks:

- Extract Microsoft results:
  - Split result of each sample based on CARO notation seen in Code snippet 4.1.
  - Extract the type of the malware from the split.
- Perform majority vote from all AV vendors:
  - Run Levanshtine ratio calculation and perform vote.
  - Extract type from the winning label.

The index of the Microsoft results is defined from the list of AV programs extracted in Code snippet 6.5. This index is used to extract the appropriate label. In order to extract the type of the malware, as detected by Microsoft, the results is split by the semicolon *:* character. As defined by CARO, the type of the malware represents the first word in the label, thus after the split, this information can be extracted from index zero.

The Levenshtein similarity ratio is implemented using a tokenizer, splitting the label in multiple words and comparing their similarity with another tokenized label from a different AV program. To better

understand this process, a simple example is depicted in Code snippet 6.6. The code makes use of a Python library called *fuzzywuzzy* which was first implemented to match different sport events over the Internet. As the intended action for this project matched the goals of the external library, it has been used successfully to denote similarities between different AV labeling techniques. The ratio threshold has been set to **50%**, denoting that at least half of the label, assuming either type of family, matches with other labels.

Code snippet 6.6: Tokenized Levenshtein distance ratio.

```
<<< from fuzzywuzzy import fuzz
<<< import Levenshtein
<<< Levenshtein.ratio("Trojan Win32","Win32 Trojan")
>>> 0.5
<<< fuzz.token_set_ratio("Trojan Win32", "Win32 Trojan")
>>> 100
```

The majority vote has been implemented as a *Counter* object, updated each time the ratio has been met. In order to improve the run-time of this module, the iterations have been cut in half. It has been done so by adding both labels to the Counter if the ratio is above the threshold keeping in mind that the ratio between *label_1* and *label_2* is exactly the same as the ratio between *label_2* and *label_1*. In order to extract the types, a list of the eight most common malware types has been generated which has been compare with the winner of the Majority vote. If one if the types in the generated list represented a substring of the label, then that type was assigned to the sample. This implementation has been chosen due to the problem of inconsistency in the labeling techniques. As most of the AV vendors do not follow the CARO syntax, extracting particular information would have been a very time consuming task. To exemplify the majority vote process, Code snippet 6.7 depicts the most important parts of the vote implementation.

Code snippet 6.7: Majority Vote and type extraction

```
        i = data['labels'][vv] # labels of current sample vv
        for j in range(0, len(data['detected'][vv])): # AV iteration
            for jj in range(j, len(data['detected'][vv])): # second AV iteration
                if not i[j] == i[jj] and not i[j] == '': # make sure is not same AV
                    if similar(i[j],i[jj]): # Tokenized Levenshtein Ratio
                        templist.append(i[j]) # If true add both to vote
                        templist.append(i[jj])
        # select most voted label
        for k,c in Counter(templist).most_common(1):
          # add result to list
            labels[vv] = k
```

After both Microsoft types and Majority Vote types have been extracted, they are inserted in MongoDB under the *labels* object for each sample. The collection containing all the behavior information is queried and through every iteration the samples are updated with both labels. In order to speed the process, the loop updating the database has been made thread ready, being able to update multiple samples at the same time, depending on the hardware capabilities. The update part of the database can be depicted in Code snippet 6.8.

Code snippet 6.8: Database update with Microsoft and Majority labels.

```
def dbThread(value,mIndex,majority,microsoft,pbar,db):
    # check if labels object is already in db
        if "labels" in value:
            return
        sampleNR = majority['samples'].index(value['virus'])
        # Majority vote type
        maj = majority['label'][sampleNR]
        # Microsoft type
        ms = microsoft['labels'][sampleNR][mIndex].split(':')[0].lower()

      # Update database based on ObjectID with correct labels
        db.APIradu.update({ '_id' : value['_id']},{'$set': {'labels': {'majority': maj, '
            microsoft': ms }}})

for value in query:
    # Select free thread from pool and start the update process
        pool.apply_async(dbThread(value,mIndex,majority,microsoft,pbar,db))
```

The Python function used for extracting labels from Microsoft, Levenshtein Majority Vote and *fuzzywuzzy* can be found on the CD in `..\Code\LabelingModule\` path.

This section has presented the design and implementation of the Majority vote. Next section will present the design and implementation of the feature extraction to be used for unsupervised learning.

6.5 Feature Extraction Module

This section includes the design and implementation of the Feature Extraction Module, where the behavior of the malicious software is extracted from the database and translated into machine learning readable format. The module follows a series of actions that is represented as a flow shown in Figure 6.4.

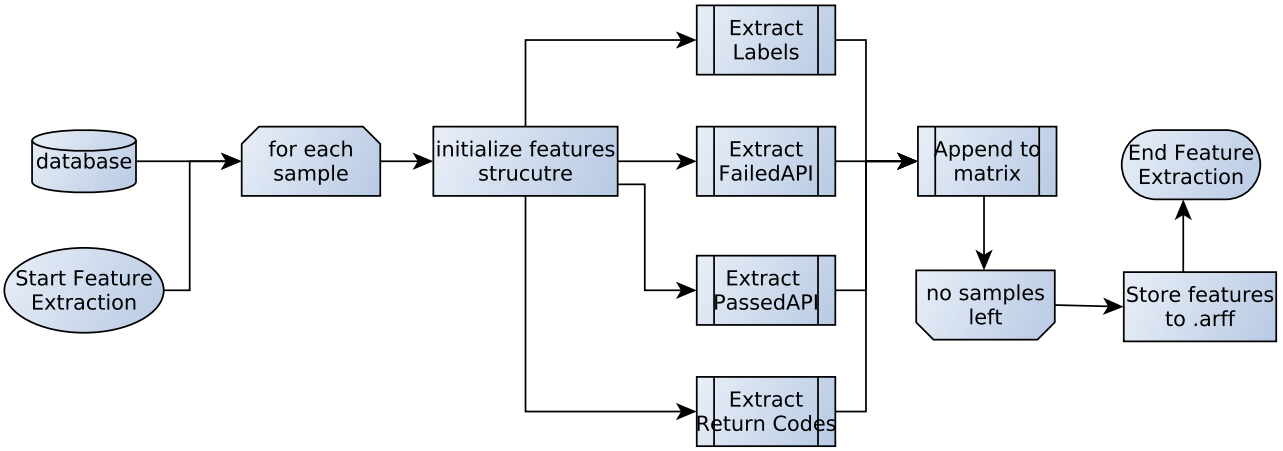


Figure 6.4: Flow chart for the Feature Extraction Module

To briefly explain the process, features are extracted from the database based on the requirements and limitations imposed by the previous sections and then stored into WEKA data-set file structure. The

execution time of this module varies from a few minutes to hours depending on the sample set in use. In order to minimize the run-time of this module, the extraction is done in multiple threads where the hardware supports it. The process starts by querying the database to retrieve all the samples available in the modified behavioral structure. Each results from the query is then analyzed extracting the features using the representation explained in Section 4.7 (p. 37). After all samples have been analyzed, the results are then saved into a WEKA readable format. The following subsections will explain detail how the process has been implemented

6.5.1 Extraction

The extraction method will be explained at a per sample level of detail. The methods and functions described in this subsections are applied on each sample retrieved from the database, in the end forming a multi-dimensional matrix consisting of:

1. Features represented by the columns of the matrix.
2. Samples represented by the rows of the matrix.

Each feature representation will have a predefined length depending on the number of distinct values seen over all samples. Thus, for the sample set used in this project, the Failed and Passed API calls will represent 157 distinct features each while the Return codes will represent a total number of 137 features. For each entry of the behavioral data, the features are extracted in parallel to minimize the number of iterations.

The process is started by simply initializing the each row, representing the sample, with the appropriate number of features as seen in Code snippet 6.9

Code snippet 6.9: Row initialization for each sample

```
# Failed,Passed API and Return Codes
# initialization
failed = [0] * (len(apiList))
passed = [0] * (len(apiList))
freturn = [0] * (len(returnList))
```

The number of processes is then depicted from the *processes* list as described in Section 4.3 (p. 20). Iterating through each API from all extracted processes the frequency of the Failed, Passed APIs and Return Codes is computed and stored at the appropriate index. This process can be seen in Code snippet 6.10. Labels are also extracted, however no computations are needed as the previous module has stored all labels in a static location. These are extracted by simply accessing the labels object with *Microsfot* and *Majority* keys.

Code snippet 6.10: Feature Extraction

```
for i in range(0, nrOfProcesses):
            nrOfAPIs = len(x['api'][i])

            for j in range(0, nrOfAPIs):
                if x['status'][i][j] == True:
                    passed[apiList.index(x['api'][i][j])] += 1
                elif x['status'][i][j] == False:
                    failed[apiList.index(x['api'][i][j])] += 1
                    freturn[returnList.index(x['return_code'][i][j])] += 1

        totalSum = sum(failed) + sum(passed)
        if totalSum < 50 and skipLowAPI:
            pbar.maxval = pbar.maxval - 1
            return

        FailedList.append(failed)
        PassedList.append(passed)
        FrequencyReturn.append(freturn)
```

In order to prevent corrupt behavioral data, samples that have less than 50 API calls, denoted by the sum of failed with passed APIs, will be excluded from the representation. Assuming that no malicious action can be done with less than 50 API calls, the excluded samples contain no information that can help the ML algorithm to better discriminate the types.

6.5.2 Storing

After all samples have been extracted and added to the temporary matrix, the data must be converted to a WEKA readable form. WEKA uses the ARFF data structure for reading sample sets which follows the syntax of a regular Comma Separated Values (CSV) file along with headers denoting the meaning of each column. To perform the conversion, the project makes use of the Python library *liac-arff*, which is able to write and read to/from ARFF formatted files. Each feature representation is stored in their appropriate ARFF file as presented in Code snippet 6.11.

Code snippet 6.11: Saving to ARFF

```
# Call to dumpArff which makes use
# of the liac-arff library
dumpArff('Representations/passedThreaded.arff',
      PassedList,
      relation="PassedAPI",
      names=list(map('Passed_{0}'.format, apiList))
    )
```

In order to distinguish between failed and passed API calls, the feature names have been modified to include the name of the representation that they belong to. The full code for the *dumpArff* function can be found in the Appendix B (p. 114).

The Feature Extraction Python implemented module,for both multi and single threaded extraction, can be found on the attached CD in `..\Code\FeatureExtractionModule\` path.

This section has described the Feature Extraction module where the behavioral data has been extracted following the specification in Section 4.7 (p. 37). Next section will present the design and implementation of the Feature Selection method.

6.6 Feature Selection Module

This section presents the design and implementation of the features selection module. It makes use of two different sub-modules in order to keep the Python implementation as simple as possible. The flow of this module is presented in Figure 6.5.

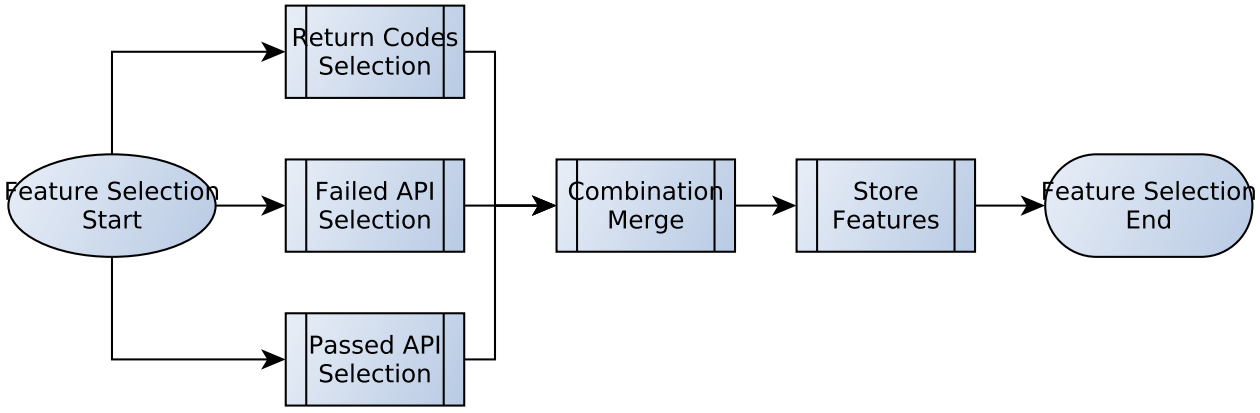


Figure 6.5: Flow chart for the Feature Selection Module

The input of this module represents the output of the Feature Extraction module. Each feature representation, denoted by the Failed API calls, Passed API calls, and Return Codes will pass through a filter for selecting the most relevant features based on their values. As discussed in Section 5.1 (p. 41), PCA was the method of choice for selecting the appropriate features based on their variance or co-variance. Both feature selection results will be saved to a final ARFF file denoted as *combinationVariance.arff* and respectively *combinationCovariane.arff*.

The module makes use of WEKA feature selection module that is called from Python using the *subprocess* library. As the size of each feature representation is significantly large, multi-threading could potentially exhaust the server memory thus, in order to prevent any unwanted errors, the module executes a single feature selection action at a time. The module executes a bash script that accepts as input a single feature representation and saves the temporary output, as seen in Code snippet 6.12.

Code snippet 6.12: Bash script for PCA feature extraction

```
#!/bin/bash

# arff input file
input=$1
# arff output file
output=$2
# Options for PCA. Used to pass co-variance option
options="$3"

# Executa command-line WEKA with
# a heap size of 6GB
java -Xmx6g -cp /home/user/weka-3-7-12/weka.jar
   weka.filters.supervised.attribute.AttributeSelection -E "weka.attributeSelection.
       PrincipalComponents
   -R 0.95 -A 1 $options"
   -S "weka.attributeSelection.Ranker -T 0 -N -1"
   -i $input -o $output
```

where,

- weka.attribute.Selection.PrincipalComponents represents the WEKA method for PCA.
  - -R 0.95 represents the amount of variance of co-variance to be chosen in dimensionality reduction.
  - -A 1 represents the number of features to combine. As the module does feature selection, combining features will end in a dimensionality reduction.
  - -C represents the centering options that will enable PCA to be computed based on co-variance instead of variance.
- weka.attributeSelection.Ranker represents the WEKA method of assigning weights to the selected features.
  - -T 0 represents the threshold of the Ranker method. It is set to zero to keep all the features that can help the ML algorithm to better distinguish between types.
  - -N -1 represents the number of features to keep. It is set to -1 to keep all the features that are above the threshold. If a number is set, for example 10, then this method will keep the first ten features with the highest statistical probability of better defining the types.

After each representation is passed through the PCA filter, the resulted selected features are then combined into a Combination representation for both variance and co-varianced based PCA. This submodule also uses WEKA and can the code can be seen in Code snippet 6.13.

Code snippet 6.13: Merging submodule of selected features

```
#!/bin/bash

# input path 1
input1=$1
# input path 2
input2=$2
# output path of merged instances
output=$3

java -Xmx6g -cp /home/user/weka-3-7-12/weka.jar
   weka.core.Instances merge
   $input1 $input2 > $output
```

WEKA function *merge* has a limit of only two inputs. As the Feature Selection module has three representations to merge, this action is split in two as follows:

1. Merge select Failed API with selected Passed API and output to a temporary ARFF file.
2. Merge the temporary ARFF file with the selected Return Codes and output to the combination representation of selected features.
3. (Optional) Remove the temporary ARFF file to save space.
4. (Optional) Merge the combination representation with labels for evaluation.

The Python functions along with the bash scripts used for Selecting Features can be found on the CD in `..\Code\FeatureSelectionModule\` path.

This section has presented the design and implementation of the Feature Selection Module. Next section will describe how the number of clusters is computed from the combination representation.

6.7 Number of clusters Module

This section describes the design and implementation of the Number of Clusters module, responsible to find the most appropriate number of clusters given a certain data-set. As the features extracted will be used to train an unsupervised algorithm, classes are not available to define the number of clusters that can be seen in the data-set. Thus, based on the data, this module will determine the optimal number of clusters using different mathematical and statistics formulas. In Figure 6.6 the flow of this module is depicted.

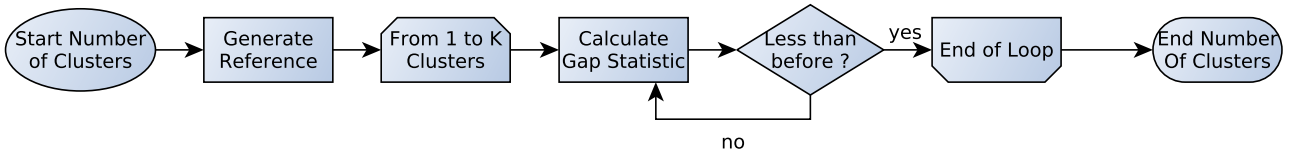


Figure 6.6: Flow chart for the Number of clusters Module

Previous discussion seen in Section 5.2 (p. 45) has determined that the best suitable method for selecting K would be using the Gap Statistic. In comparison with other methods, the Gap Statistic can be automatized to detected the number of clusters. To do so, a random distribution is used to compare its log likelihood with the log likelihood of the data by increasing the number of clusters. It is assumed that the values will increase until a break point after which the values will start decreasing. The corresponding number of clusters where the break point occurs is defined to be the optimal K for the data-set at hand.

In order to implement this module, the use of `scikit-learn`, `numpy` and `scipy` libraries is essential for creating the random distribution as well as for computing the log-likelihood of clusters using multi-threaded kmeans++. The values of the reference distribution are bound by the minimum and maximum values seen in the the actual data, as seen in Code snippet 6.14. The implementation is based on the work presented in [Vejdemo-Johansson, 2013].

Code snippet 6.14: Generating a random distribution

```
# Create reference distribution
shape = data.shape
tops = data.max(axis=0)
bots = data.min(axis=0)
dists = scipy.matrix(scipy.diag(tops-bots))
rands = scipy.random.random_sample(size=(shape[0],shape[1],1))
```

After the random distribution has been generated, the module will loop through a range of clusters. The log likelihood is calculated for both data-sets and gap statistic is calculated using the equation depicted in Section 5.2 (p. 45). The module stores all values in a list, comparing the currently calculated value with the previous one in order to decide if the loop continues or breaks. The resulting value of K will be used for evaluation purposes or for assigning the number of clusters with algorithms that require such an input. The implementation of the Gap statistic can be seen in Code snippet 6.15,[Vejdemo-Johansson, 2013].

Code snippet 6.15: Gap Statistic calculation for a range of K

```
for k in maxK:
        # Cluster the data with k clusters
        (kmc,kml) = k_means(data, k, init='k-means++', n_jobs=8)
        # calculate dispersion matrix for each cluster for the data-set
        disp = sum([dst(data[m,:],kmc[kml[m],:]) for m in range(shape[0])])
        # Cluster the reference with k clusters
        (kmc,kml) = k_means(rands[:,:,0], k, init='k-means++',n_jobs=8)
        # calculate dispersion matrix for each cluster in reference data-set
        refdisp = sum([dst(rands[m,:,0],kmc[kml[m],:]) for m in range(shape[0])])
        # calculate Gap Statistic based on the values got from data and refdisp
        gaps[k] = scipy.log(refdisp)-scipy.log(disp)-np.std(refdisp)
```

The implementation of the Python Gap Statistics can be found on the CD in `..\Code\NumberOfClustersModule\` path.

This section has presented the design and implementation of the module that is selecting the appropriate number of clusters. Next section will focus on the design and implementation of the Machine Learning module.

6.8 Unsupervised Learning Module

The section presents the design and implementation of the Unsupervised Machine Learning module where the output of previous modules is used the train a selected algorithm and evaluate its results based on some predefined labels. This module follows the flow of Figure 6.7.

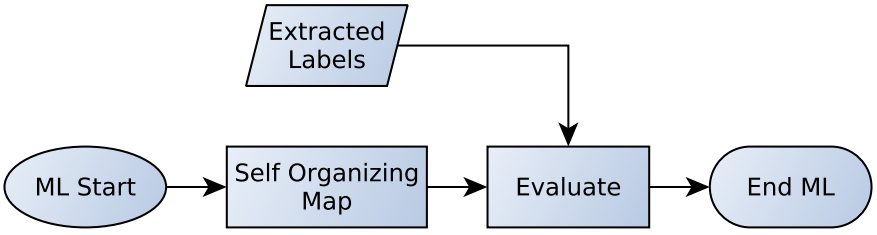


Figure 6.7: Flow chart for the Feature Selection Module

This module relies on the output presented by the Feature Selection module as well as on the output of the Labeling Module. The ARFF file containing the features is passed to SOM algorithm in WEKA and the file containing the labels is used for evaluation after the clustering has been done. Code snippet 6.16 exemplifies the code used for this module.

Code snippet 6.16: Call function for SOM algorithm

```
1 #!/bin/bash
2
3 java -Xmx14g -cp /home/user/weka-3-7-12/weka.jar weka.Run SelfOrganizingMap
4        -L 1.0 -O 2000 -C 1000 -H $height -W $width -c last
5        -t $input > $output
```

where,

- -L represents the learning rate of the algorithm.
- -O 2000 represents the number of ordering epochs. Also can be defined as number of iterations.
- -C 1000 represents the number of calculation iterations.
- -H x -W y represents the grid size of x by y. This number has been set as it represents a fair grid size given the number of existing malware types. It gives the opportunity to observe in detail how each cluster relates to its clusters.
- -c last represents the class index. WEKA supports numeric indexes or the *first* and *last* defined as the first and respectively last index.

The output of the algorithm consists of the following information:

- Cluster names based on the provided classes.
- Incorrectly classified instances based on the majority vote in previous point.
- Evaluation matrix defining the instances assigned to each created cluster.

The code for calling Self Organizing Map from WEKA within Python can be found on the CD in `..\Code\MachineLearningModule\` path. There exist two bash scripts that evaluate and output the sample set with the extracted clusters as classes for future classification use.

The information provided by this module will be used in Chapter 7 to analyze and evaluate the performance of the used algorithm. This chapter firstly discussed the choice of programming language and provided an overall flowchart of the programming modules available in the system. The design and implementation of each module has been explained in detail in Section 6.3 (p. 66), Section 6.4 (p. 68), Section 6.5 (p. 71), Section 6.6 (p. 74) and Section 6.7 (p. 76). Note that the code and result files for each module are fully available on the attached CD.

INTENTIONALLY LEFT BLANK

Part IV

Results

Chapter 7

Evaluation

This chpater will present the results of the evaluation done on clusters generated using Self Organizing Map. A comparison will be done between the resulted number of clusters from both SOM and Gap Statistics, and the number of distinct types, Majority or Microsoft, extracted from the data-set. The intent of this evaluation is to observe the differences seen between labels, Gap Statistics and the chosen algorithm in order to determine if a link exists between the collected behavioral data and labels created using static analysis. The chapter is split accordingly in three sections denoting the Label Evaluation in Section 7.1 (p. 83), Feature Contribution in Section 7.2 (p. 87) and SOM Evaluation in Section 7.3 (p. 88).

In order to fairly evaluate the extracted data, the ARFF files have been limited to contain a uniformly distributed number of samples in terms of Microsoft labeling. Table 7.1 presents the values seen for the five most common types used with this representation.

Type	**Number of Samples**
Clean	52777
Trojan	35227
Backdoor	24056
Trojan Downloader	23543
Trojan PWS	20550

Table 7.1: Filtering methods comparison

The Majority Vote will include the same sub-set of sample seen in the Microsoft labeling, however the distribution is not uniform, where approximately three fourths of the samples being part of the Trojan class. Table 7.2 presents the value seen for the Majority Vote. It has to be mentioned that even though the labels differ, the sample-sets contain the same behavioral data. It is known that the non-uniformity of the data might indeed return more favorable results however, the intent of this evaluation was to compare the two labeling techniques and determine if the behavioral data contains information about AV extracted malware types.

From the start, the number of distinct labels can be seen between Microsoft and Majority labels. However, the evaluation focuses on defining the differences between these labeling techniques and the number of clusters that can be achieved using:

- Gap Statistics selected in Section 5.1 (p. 41).
- Self Organizing Map selected in Section 5.4 (p. 51).

Type	Number of Samples
Trojan	124559
Backdoor	18716
Adware	8253
Malware	2172
Worm	1620
Virus	645
Rootkit	181
Spyware	7

Table 7.2: Filtering methods comparison

7.1 Label Evaluation

The Gap Statistic is used estimate the number of clusters that corresponds to the data-set in use. It uses a reference distribution with no obvious number of clusters and calculates the log-likelihood using a range of clusters. As this method uses kmeans++ to cluster and evaluate the log-likelihood, its results are expected to differ from SOM algorithm. Comparing the two based on how the clusters are created, k-means++ uses centroids that cannot always take the shape of the data while SOM generates the clusters based on geometrical calculations, being able to better fold on the input vectors. Due to this limitation of kmeans++, the clusterer is more prone to noise. After running the experiment on all four feature representations, the results can be seen in Figure 7.1.

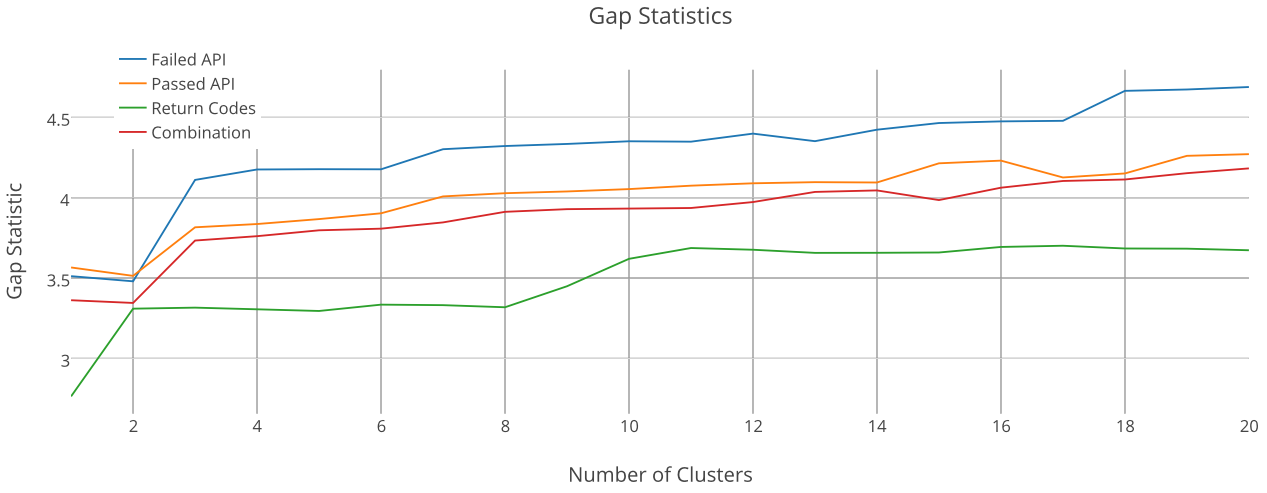


Figure 7.1: Gap statistics for multiple number of clusters using 4 different feature representations.

From the Gap Statistic results, it can be seen that the log-likelihood does not reach a maximum value point in neither one of the feature representations. Instead the values keep increasing up to 20 clusters where the experiment has stopped due to long running times. The algorithm has chosen to use 14 clusters based on the Combination representation where there exists a slight decrease in the likelihood between cluster 14 and 15. This suggests that 14 clusters better define the data-set in comparison with 15 clusters.

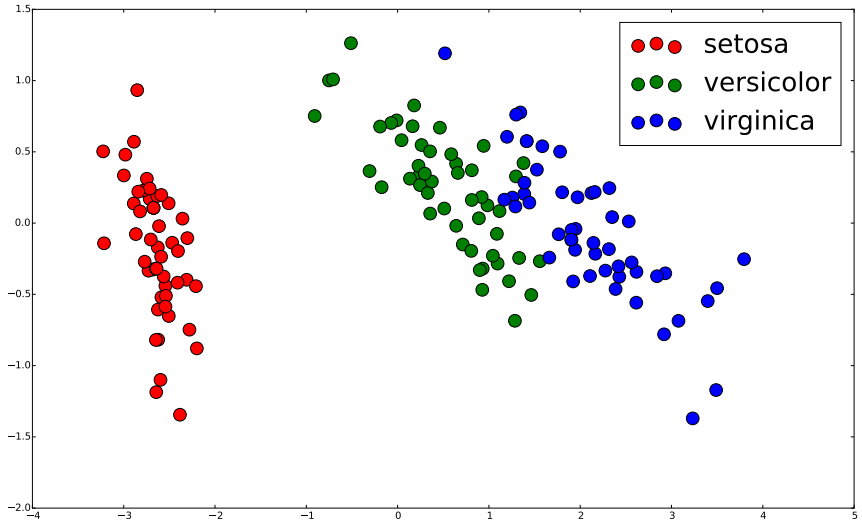
As described in previous paragraph, the Gap Statistic makes use of kmeans++ to determine the log-likelihood of the data falling into a range of clusters, representing a linear discrimination and thus limiting the shape of the clusters. On the other hand SOM projects the data to a two-dimensional space while maintaining the properties of the high-dimensional data, allowing the clusters to take any shape, depending on the values they contain. As both Gap Statistics and SOM do not make use of the labels presented in Table 7.1 and Table 7.2 for determining the clusters, they will be used for comparison with already known data. After running the experiment for determining the number of clusters, the results can be seen in Table 7.3.

Method	Options	Clusters
Microsoft	N/A	5
Majority	N/A	8
Gap Statistics	N/A	14
SOM	2x2	4
SOM	1x5	5
SOM	3x3	9
SOM	4x4	16

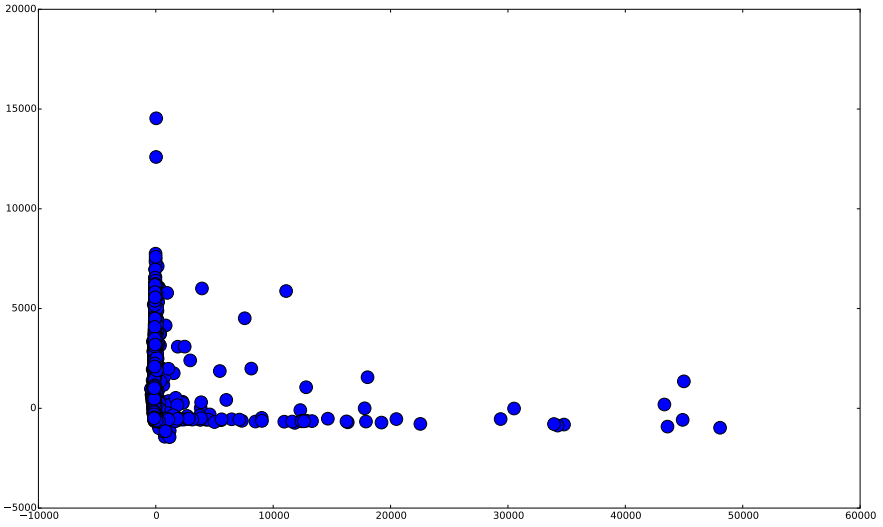
Table 7.3: Results of number of clusters using Gap Statistic and different configurations of SOM.

Results from Table 7.3 are based on the statistics calculated on 33 co-variance chosen features as they provide the best suitable solution in terms of run-time. Analyzing the results, none of the methods seem to provide the same information as the labels extracted from the data. The number of clusters selected by Gap Statistic suggests that the behavior of the sampled malicious software can be grouped in many more clusters than suggested by both Microsoft and Majority labels. Furthermore, the SOM clusterer seems to define the number of clusters equal to the size of the grid. The evaluation of clusters for SOM has stopped at the grid size of 4x4 due to long running times of the algorithm. The current grid size has clustered the provided data-set in approximately 24 hours. The 5x5 grid size has been also attempted however the clusterer did not converge after two days and thus the analysis was not continued.

In order to understand the extent of selecting the optimal number of clusters when dealing with malicious software, a side-by-side comparison is presented in Figure 7.2 (p. 85) between the well-defined IRIS data-set and the current data-set used in this project. It has to be noted that the figure presents the data-sets in a 2-Dimensional form, transformed using PCA.



(a) IRIS Dataset



(b) Malware Dataset

Figure 7.2: Comparison between IRIS and Malware datasets.

The difference between the two sample sets is visible directly from the graphs. The IRIS example has three well-separated groups of values that can be easily recognized by both SOM and Gap Statistics. This is mostly due to the large inter-cluster distances and small intra cluster distances, making this data-set the best case scenario for every clustering algorithm. Observing the subset of the malware behavior graph and comparing it to the IRIS graph, it can be noticed that there is no obvious grouping that can be done. Any intra or inter-cluster statistics will yield information that is ambiguous for most clustering algorithm and this can be seen in the calculations of the Gap Statistics.

Based on the malware data-set in use, Gap Statistic calculates the number of clusters to be 14 due to the drop in the log-likelihood between cluster 15 and 14, as seen in Figure 7.1 (p. 83). This value has been chose even though the values continue to increase after cluster 15. As maybe this does not provide the appropriate number of clusters, it can be denoted that there is a significant difference between the number of labels provided by both Majority vote and Microsoft compared with the number of clusters evaluated by Gap Statistics. As a reference distribution is used where it is assumed that the number if clusters represents the number of points, the log-likelihood of the clustered data will increase suggesting that the appropriate number has not yet been reached.

In order to choose the best suitable option for SOM algorithm, the clustering results have been evaluated based on both labeling techniques. The grid sizes of the Self Organizing Map that have performed better in correctly assigning labels to the created clusters will be selected for further analysis. It has to be noted that, the percentage presented in Table 7.4 illustrates the dissimilarity of the clustered data with the labels provided from various AV vendors where a lower value is preferred, see Section 5.5 (p. 56). This can be used to determine if the generated clusters match the extracted AV labels.

Method	Options	Clusters	Incorrectly Clustered Instances	
			Majority	Microsoft
Kmeans++	N/A	5	43.5195%	69.4005%
Kmeans++	N/A	8	44.4987%	73.1257%
Kmeans++	N/A	14	45.1224%	73.3319%
Kmeans++	N/A	16	45.9152%	73.0008%
SOM	2x2	4	21.1472%	65.3255%
SOM	1x5	5	21.2569 %	66.8441%
SOM	2x4	8	29.4634%	69.4325%
SOM	2x7	14	30.1255%	70.2223%
SOM	4x4	16	30.4701%	71.8578%

Table 7.4: Clustering Methods Compared.

Table 7.4 provides information about how labels fit the generated clusters when using kmeans++ and Self Organizing Map. The grid sizes and the number of clusters chosen for evaluation are based on the distinct number of types seen in both Microsoft and Majority labels, Gap Statistics and a fair assumption for the grid sizes that closely match the determined values for a two-dimensional map. The incorrectly clustered instances are presented for both Microsoft and Majority Vote labels in order to determine if there is a link between them and the created clusters.

Following the results it can be denoted that kmeans++ created behavioral clusters do not match the information extracted from labels. On the other hand Self Organizing Map seems to better define the clusters, observing an increase in correctly classified clusters by approximately 25%. As previously discussed, this can be due to the fact that the projection created on the two-dimensional map succeeds at preserving the topological properties of the selected features.

As SOM provides better accuracy based on the Majority labels, the Evaluation matrix and the created map will be analyzed in detail. The goal of the evaluation represents an attempt at understanding what the clusters hold and how closely are they related with AV extracted types. Next section will evaluate the contribution of selected features in correctly clustering the data.

7.2 Feature Contribution

This section aims at providing information about which features can better describe the Majority labels extracted using Levensthein distance from multiple AV vendors. The following tests will go through all feature representations that were selected using co-variance PCA. A range of grid sizes will be tested after which the best ones will be selected in order to be analyzed in detail. The results can be seen in Table 7.5.

SOM Options / Features	2x2	2x4	3x3
Failed API	33.6561%	33.8341%	32.9094%
Passed API	20.8789%	25.3533%	25.6454%
Return Codes	32.1217%	34.4720%	35.1104%
Combination	21.1472%	29.4634%	30.4701%

Table 7.5: Incorrectly classified instances for multiple feature representations.

From Table 7.5, it can be seen that the Passed API features provide the best accuracy when evaluated on the Majority Vote. The information within the Passed API representation contains the frequency of calls that have succeeded on the infected machine. Even though the accuracy of the evaluation done on this representation represents the best result for all tested grid sizes of SOM, a more detailed behavioral representation is preferred. As most of the malicious programs do not have information about the contents of the infected machine, a large number of API calls are initiated to exhaustively search the infected machine. This will lead to a large number of Failed API calls along with their respective Return Codes. The focus of the Combination matrix is to include all actions of the malware, either if the actions have succeeded or not.

The combination representation contains a better description of the malware behavior and, even though its accuracy is below the one of Passed API, the created clusters are better defined. Confirming this on a 2x2 grid size, the following information is denoted:

- Passed APIs create 4 unbalanced clusters where approximately 99.5% of the instance fall into a single cluster. The other three clusters hold the rest of 0.5% of the data. As most of the samples are labeled as Trojan, the evaluation algorithm assigns the Trojan label to the clusters containing most of the instances. This results in higher correctly classified instances due to the unbalanced distribution of types.

- The Combination representation also creates 4 clusters where approximately 90% of the instances fall into a single cluster. The rest of the instances are evenly spread among the other three clusters. As with the Passed API, the accuracy of the evaluation is mostly define by the Trojan.

Following the above remarks it can be concluded that the combination representation tends to describe a more detailed cluster assignment as it contains more information about the behavior of the malware. In order to better analyze the clusters, in the next section, the combination representation has been chosen to further inspect the assignment of clusters for a various number of grid sizes.

7.3 Algorithm Evaluation

This section presents the clustering results from SOM evaluated on the Combination of features selected in previous section. Different options of the clustering algorithm will be discussed in terms of cluster distribution created by the map, as well as in terms of the Evaluation matrix. By the end of this section, it will be decided if it is sufficient to use labels provided by AV vendors in order to assign types to behavioral data or if other methods should be researched. Furthermore a cluster-based classification will be tested to demonstrate the accuracy increase of classifying using well-defined classes that are built from the behavioral data and not provided by AV vendors. The following subsections will go through the options of the Self Organizing Map using 33 PCA features based on co-variance.

7.3.1 2 by 2 Grid

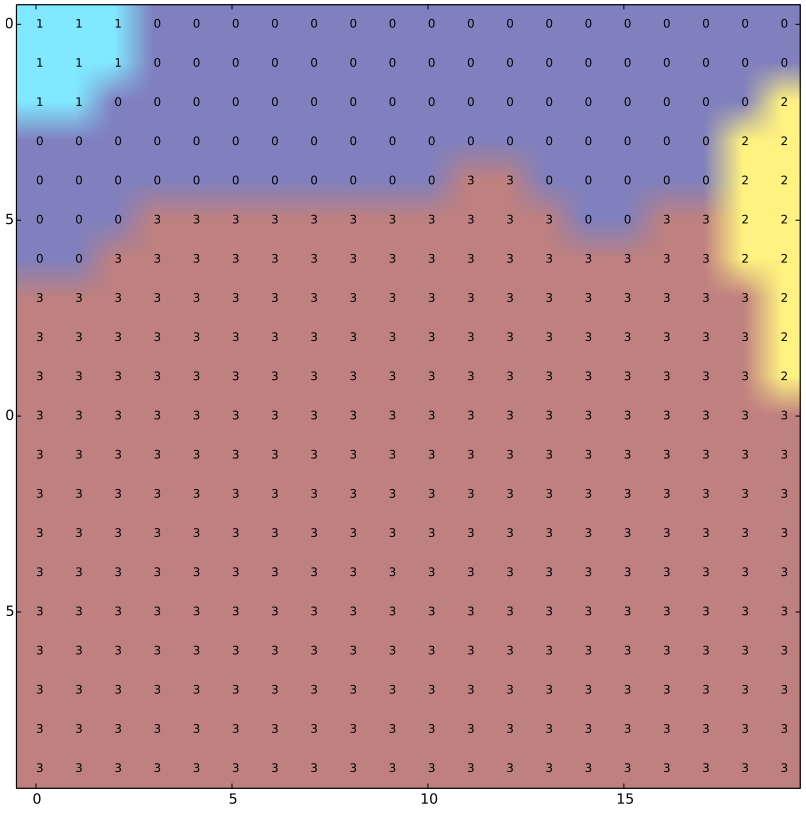


Figure 7.3: Visualization of clusters on a 2x2 grid.

The 2x2 SOM algorithm generates four clusters based on the behavioral data. The evaluation done on the Majority Vote labels shows a 79% correct assignment. The Trojan type is mainly responsible for the high percentage as it represents the majority of the samples. Even though this type represents a broad spectrum of malware, it can also contain other malware inside. However due to the project limitations in terms of virtualization and Internet access, these sub-types may or may not be present in the behavioral data and thus already introducing a gap between the labels and collected data. Spyware, Rootkit, Virus and Worm represent the bottom-line in terms of samples, from the Majority Vote. Their definition from Section 2.2.2 (p. 8) suggests that based on their intended actions they will exhibit different behaviors. The results shown in Table 7.6 show that there is no discrimination between these types and Trojan where almost all instances are assigned to the aforementioned cluster.

However, by analyzing the created SOM map, it achieves the shape of global malware distribution, where approximately 71% of malware are Trojans, 16% are Viruses, 8% Worms and the rest 1%, representing other malicious software, see [BullGuard, 2015]. This assumption cannot be verified unless extensive analysis is done on each created cluster.

This leads to the conclusion that the information stored in the clusters created by the SOM algorithm using 2x2 grid on the feature combination does not contain the types of malware extracted from the AV vendors. The Trojan instances are mostly the ones which spread in different groups along with a small portion of Adware and Backdoor instances. It can be seen that generated clusters do not follow distribution of the Majority Vote nor Microsoft labels. Due to the small grid size it can be assumed that the clusters contain a generalized description of the malware behavior based on the selected features. Future research must be done for each cluster in order to determine what malicious actions the samples within are taking. The 2x2 SOM has also been evaluated on labels using families. The results can be seen in appendix G (p. 119).

Cluster / Labels	C1 - N	C2 - W	C3 - B	C4 - T
Trojan	432	749	391	122987
Backdoor	1	43	140	18532
Adware	30	0	15	8238
Malware	0	1	3	2168
Worm	1	4	10	1605
Virus	0	0	0	645
Rootkit	0	0	3	178
Spyware	0	0	0	7

Table 7.6: Evaluation matrix for 2x2 SOM grid on Majority Labels.

7.3.2 1 by 5 Grid

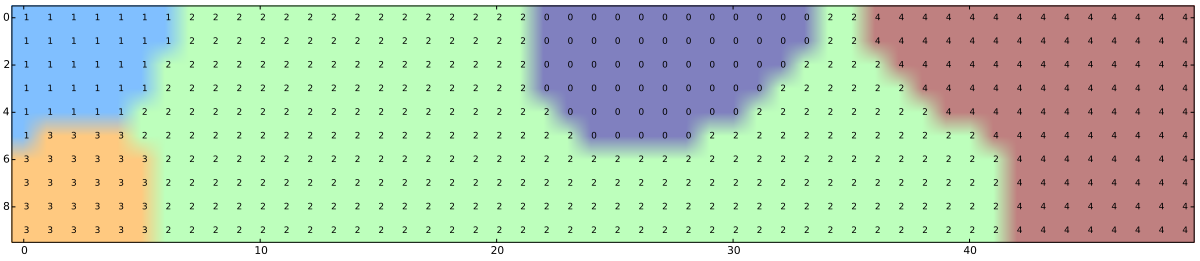


Figure 7.4: Visualization of clusters on a 1x5 grid.

The 1x5 SOM grid size has generated five clusters based on the behavioral data and matches the number of distinct types seen in the Microsoft labeling. By observing Figure 7.4 and analyzing Table 7.7 it can be denoted that the majority of instances assigned to a single cluster has not changed in a significant way in comparison with the 2x2 grid. Comparing the two evaluation matrices it can be seen that 237 samples from the Trojan cluster have been assigned to the newly created cluster. A possible explanation could be that, as a new node has been introduced, the 237 samples have chosen it as their BMU and modified its weight, moving it away from the node defining cluster number four containing most of the instances.

Based on the evaluation, and assuming that the labels are correct, the behavior clusters do not define the extracted AV types of the malware as they do not match neither of the distributions seen in Microsoft or Majority vote labeling techniques. However it can be seen that as the number of nodes is increasing, the cluster containing the majority of the samples is divining in smaller groups resulting in a better discrimination of the behavior data by the SOM algorithm.

Cluster / Labels	C1 - N	C2 - A	C3 - W	C4 - T	C4 - B
Trojan	432	237	748	122751	391
Backdoor	1	17	43	18515	140
Adware	0	35	0	8203	15
Malware	0	4	1	2164	3
Worm	1	2	4	1603	10
Virus	0	1	0	644	0
Rootkit	0	1	0	177	3
Spyware	0	0	0	7	0

Table 7.7: Evaluation matrix for 1x5 SOM grid on Majority Labels.

7.3.3 2 by 4 Grid

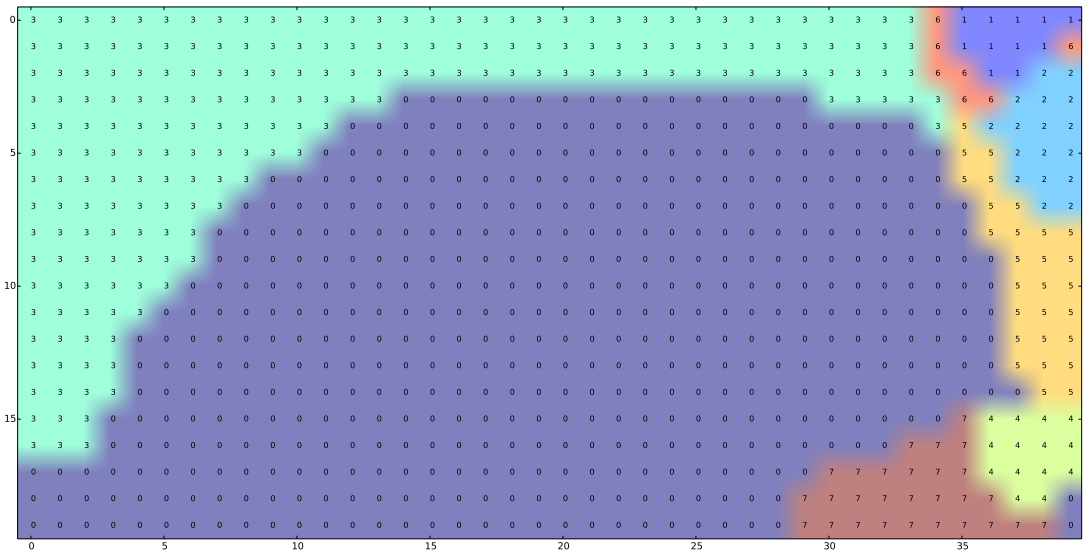


Figure 7.5: Visualization of clusters on a 2x4 grid.

The 2x4 SOM algorithm generates eight clusters based on the presented behavioral data and matches the number of distinct types present in the Majority Vote. The evaluation done on these labels shows an accuracy of 71%. Comparing with the previous representation on a 2x2 grid, the results seem to have the same shape, where the instances are mostly assigned to the Trojan cluster, with a decrease of approximately 30,000 samples. However as the algorithm is non-deterministic it is believed that the created maps cannot be directly correlated in terms of the location of the clusters. This is due to the fact that the weights of the nodes are randomized and no pseudo-random algorithm is applied in order to assign the same starting weights for different runs. Even though based on Figure 7.5, the shape of the map follows the same pattern, it can be noticed that different groups tend to emerge from the previously created clusters. Table 7.8 sustains the idea by observing the values of cluster six, where a small portion of Trojan, Backdoor and Adware are assigned to it. Nevertheless the malicious types consisting of Spyware and Rootkit still fall in the Trojan majority cluster, even though their definitions state that they perform unique tasks that should discriminate them from other types, as described in Section 2.2.2 (p. 8). On the other hand, Virus and Worm have split to different clusters denoting that a more detailed clustering is done with this chosen grid size and differences in behavior can be better discriminated.

As more clusters have been created it is safe to assume the this grid size can cluster the behavior of malware in a more detailed manner. Even with this increase in the level of detail, the information within these clusters do not seem to hold the AV defined types of the malware. It can be discussed that the values seen for the selected 33 features are close together making the discrimination between similar groups a hard task to achieve.

Cluster / Labels	C1 - N	C2 - W	C3 - M	C4 - T	C5 - N	C6 - A	C7 - B	C8 - N
Trojan	701	126	236	109573	406	12608	391	518
Backdoor	42	4	17	17995	1	494	140	23
Adware	0	0	35	7774	0	415	15	14
Malware	1	2	4	2092	0	69	3	1
Worm	4	13	2	1471	0	115	10	5
Virus	0	0	1	513	0	131	0	0
Rootkit	0	0	1	174	0	3	3	0
Spyware	0	0	0	7	0	0	0	0

Table 7.8: Evaluation matrix for 2x4 SOM grid on Majority Labels.

7.3.4 4 by 4 Grid

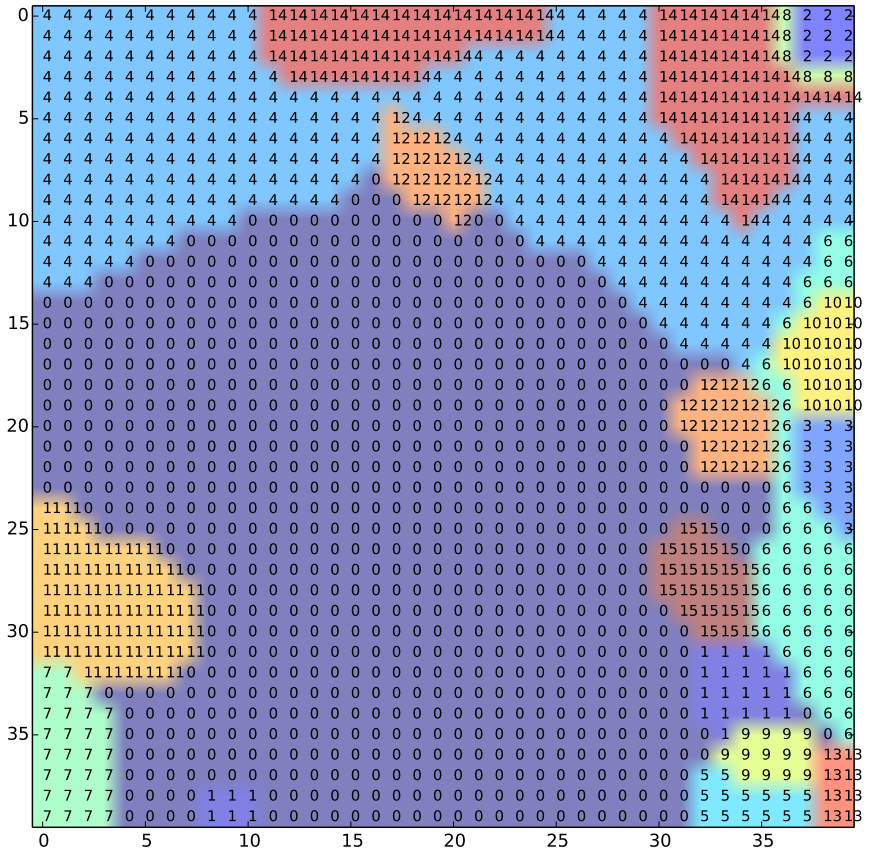


Figure 7.6: Visualization of clusters on a 4x4 grid.

The 4x4 SOM algorithm generates 16 clusters based on the presented behavioral data and represents a close match to the Gap Statistics calculation. The evaluation done on the Majority Vote shows an accuracy of 69% representing the lowest value in comparison with previously tested grid sizes. However the difference between previous tested grid sizes is not that noticeable given the fact that the number of clusters has doubled. Even though it is clear that the accuracy of the evaluation tends to decrease as the number of clusters increases, the amount of information that can be extracted from the malware behavior increases considerably. The cluster previously containing a large number of Trojans, is now split into multiple clusters containing similar behaviors based on the selected features. The Trojan accuracy has dropped but at the same time, the accuracy of Backdoor has increased. Due to the number of clusters, the evaluation matrix seen in Table 7.9, contains only the 8 most important clusters to evaluate. The full evaluation matrix for this grid size has been included in Appendix H (p. 120). The number of instances tend to split from the cluster holding the majority of samples. This denotes that, for this grid size, the behavior of the malware is defined in more detail allowing for new clusters to emerge.

Cluster / Labels	C1 - N	C2 - A	···	C5 - B	···	C13 - T	···
Trojan	381	21384	···	1406	···	97210	···
Backdoor	1	462	···	239	···	17542	···
Adware	0	2237	···	19	···	5889	···
Malware	0	150	···	20	···	1976	···
Worm	0	146	···	23	···	1390	···
Virus	0	297	···	1	···	343	···
Rootkit	0	5	···	0	···	170	···
Spyware	0	2	···	0	···	5	···

Table 7.9: Evaluation matrix for 4x4 SOM grid on Majority Labels.

It is believed that as the grid size increases, more small groups will emerge from the Trojan cluster as the SOM algorithm will be able to define the behavior in more detail. Overall it can be concluded that the selected features do not exhibit the behavior of types as defined by Microsoft or by the Majority Vote done on multiple AV vendors. It can be discussed that classifications based on AV labels, done in previous research papers cannot be validated based on the results shown in this paper. Even though, in some cases, high accuracy can be achieved, the algorithm was forced to learn the structures of behavioral data from static generated labels that present no similarities.

This section has described the evaluation done on multiple SOM grid sizes. The next section will introduce the idea of cluster-based classification, where, using a proof of concept, it is shown how the accuracy of classification can achieve high discriminating performance if the labels are built from the behavioral data.

7.4 Cluster-based Classification

Concluded in previous section, the generated clusters do not exhibit the behavior of AV generated malware types but they still contain information about some behavior of the malicious software. This section proposes a new technique of classification based on clusters created using Self Organizing Map. The clustering techniques used can improve the analysis of malware behavior by clustering the data based on similar behavior. Samples from the created clusters can then be further analyzed to understand their purpose by exploring patterns of selected features for each generated cluster. If such definitions can be created, the clustering done in this project can have great implications when classification is applied. The steps for improving classification using clustering can be described as follows

1. Extract features describing the behavior of the malicious program as presented in Section 2.2.2 (p. 8).
2. Cluster the data selecting a grid size based on the required level of detail. A smaller grid size will result in more general clusters while an increase in the grid size will result in capturing more detailed clusters as concluded in Section 7.3 (p. 88).
3. Analyze each created cluster and assign labels based on the information they contain. This could potentially improve the performance of malware analyzing in terms of time required to statically analyze samples.
4. Train a supervised algorithm using the data and labels generated from the created clusters.
5. Classify new malware based on their behavior and assign appropriate classes.
6. Repeat step 5 for a determine number of samples. Regenerate the clusters by going to step 2 after a defined period of time or after a defined number of samples.

In order to exemplify such a technique and demonstrate the increase in accuracy based on the collected behavioral data, a short example will be presented where the data-set using Microsoft Labels will be trained using Random Forests and evaluated based on a train/test split. As it is out of the scope in this project to determine the best classification algorithm and its options, Random Forests with 160 trees will be used as the Machine Learning algorithm based on the research done in our previous work, [Pirscoveanu et al., 2015]. The evaluation techniques for classification methods differ from the ones used for clustering. Thus, the AUC value, F1-Measure, Precision, Recall, True Positive and Negative rates along with confusion matrices, as described in [Pirscoveanu et al., 2015], will be used to determine the level of improvement between using classification on the labels provided by AV vendors and labels provided by clustering the behavioral data.

Microsoft Classification

In order to perform classification and evaluate the results, the data-set using Microsoft labels has been evenly split into 66% training and 34% testing, where both subsets contain a fair number of samples from each label. After the split, the training set is used to construct the model using RF algorithm with 160 trees and the testing set is used to evaluate the classifier. The first classification test results can be seen in Figure 7.7.

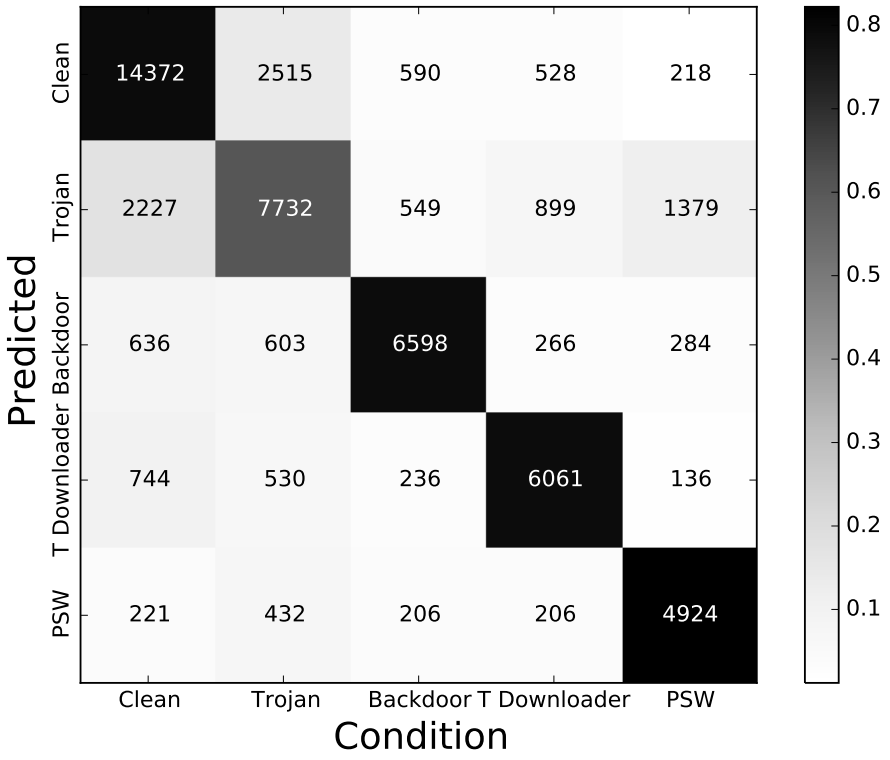


Figure 7.7: Confusion matrix using Microsoft Labels classification

The diagonal of the Confusion matrix defines the number of True Positives, the rows define the False Positives and the columns the False Negatives. Even though the samples are uniformly distributed based on Microsoft Labeling it can be seen that the algorithm fails to capture the behavior based on the provided AV labels. Trojan represents the worst classification with more than 7,000 FPs and FNs which represents approximately 50% classification accuracy. Other types are better classified however, the Clean type represents samples that have not been detected as malware. As discussed at the beginning of the project, Microsoft has a detection rate of 80%, which forces the classification to be incorrect given that all the samples are known to be malicious programs. The evaluation results can be seen in Table 7.10.

A few problems that may occur when classifying types using labels denoted from static analysis and behavioral data collected using dynamic analysis could be:

- Similar values denoted from the behavioral data can have different labels forcing the classification to randomize its choice if a decisive vote cannot be made.
- Labels could be created using different properties of the data which could not be compatible with the behavioral data, as concluded in Section 7.3 (p. 88).

Class \ Eval.	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area
clean	0.790	0.110	0.789	0.790	0.789	0.937
trojan	0.655	0.122	0.605	0.655	0.629	0.896
backdoor	0.807	0.040	0.787	0.807	0.797	0.965
trojandownloader	0.761	0.036	0.786	0.761	0.774	0.955
pws	0.709	0.023	0.822	0.709	0.762	0.967
Weighted Avg.	0.748	0.080	0.751	0.748	0.749	0.939

Table 7.10: Evaluation done on Classifying based on Microsoft Labels.

Evaluating in more detail the results of the classification seen in Table 7.10 it can be observed that the values of the False Positives if fairly large consolidating the statements made in previous paragraph. Even with high values of FPR and low values of Precision the AUC values suggests that there is a fair discrimination between types which somehow contradicts the large number of false positives seen in the evaluation.

Defining that it might not be the best approach to classify behavioral data using AV generated malware types, the project proposes a cluster-based classification approach where the labels of the classification are define by the clusters created. Figure 7.8 presents the confusion matrix of the classification done using RF on the created clusters using SOM algorithm on a 1x5 grid size. The classes have been named accordingly to the cluster assigned in the unsupervised learning.

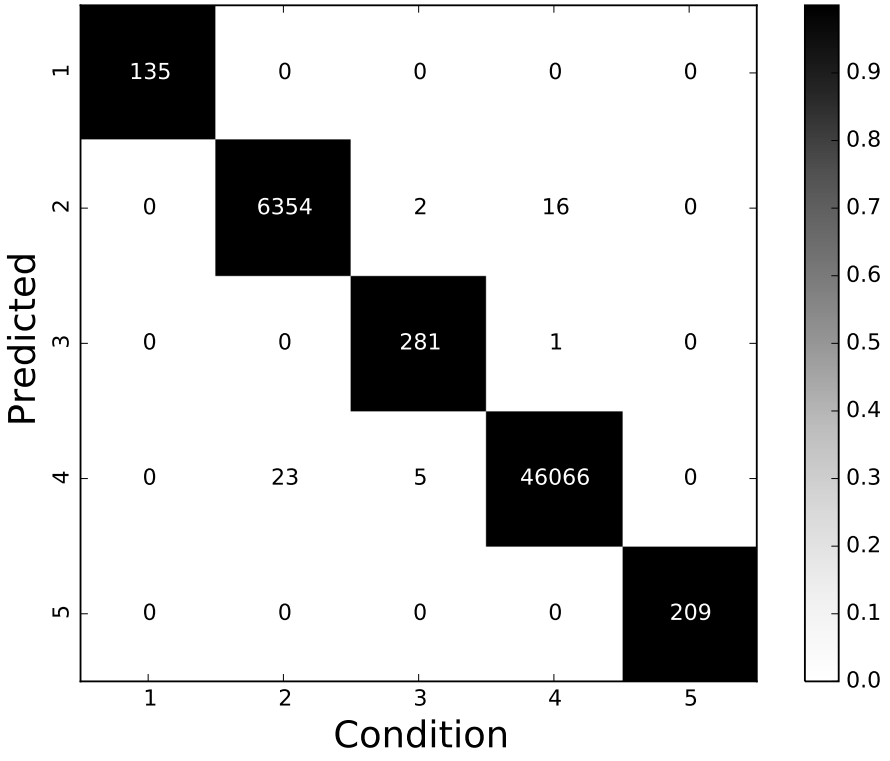


Figure 7.8: Confusion matrix using cluster-based classification on 5 clusters

It can bee seen that even though most of the class have very low samples in comparison with class number four, they are correctly classified with very low or no FPs and FNs. This claim is backed up by Table 7.11 where it can be seen that the classifier performs close to perfection in every aspect. It can be discussed that if a larger grid size of SOM was used when classifying the clusters, the classification would perform even better as the clusters would describe a more detailed representation of the behavior. This denotes that supervised algorithms can correctly classify data with low samples if the labels are correctly defined. The only challenge that this approach provides is the lack of created class information, which denotes that further research must be done in order to determine the meaning of each cluster. The purpose of the malware assigned to a particular class can be extracted by carefully analyzing the features that contributed to creating the cluster.

Eval. / Class	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area
cluster1	1.000	0.000	1.000	1.000	1.000	1.000
cluster2	0.996	0.000	0.997	0.996	0.997	1.000
cluster3	0.976	0.000	0.996	0.976	0.986	1.000
cluster4	1.000	0.004	0.999	1.000	1.000	1.000
cluster5	1.000	0.000	1.000	1.000	1.000	1.000
Weighted Avg.	0.999	0.004	0.999	0.999	0.999	1.000

Table 7.11: Evaluation done on cluster-based classification.

This section has presented the proposed approach of cluster based classification where it has been shown, using proof-of-concept that classifications done on cluster created classes can better discriminate between similar groups. The next section will present the discussions that may arise from using this approach.

Before starting a discussion on ways that the project can improve the clustering of malware behavior, a short summary of the conclusions depicted in this chapter are listed below:

- The number of clusters resulted from Gap Statistics and different grid sizes of Self Organizing Map suggested that there is no link between the labels generated by AV vendors and the groups created by the unsupervised Machine Learning algorithms.
- The information withing the Combination matrix provides a more detailed representation of malware behavior despite the lower accuracy when evaluated using Self Organizing Map on Majority Vote labels.
- After evaluating the labels using various SOM grid sizes it has been concluded that no link is present between the AV generated labels and the behavioral data. However, the SOM algorithm is able to distinguish between different malware behaviors as the number of nodes used increases.
- Having decided that no link exists between AV labels and malware behavioral data, the project has proposed a cluster-based classification approach where labels are generated based on created clusters. It has been concluded that classification can be improved if the classes are well defined and describe the data.

7.5 Discussion

This sections contains discussions to highlighted problems in the Evaluation Chapter. Furthermore, it will be discussed how the project can be improved in future-work by presenting a complete behavioral based system that can detect and group malware using only behavioral data.

It can be discussed that based on the results depicted in this chapter, the clusters do not discriminate between malicious types provided by AV vendors possibly due to:

- **Selected Features** - other feature representations should be researched that may better describe the types. A potential solution emerges by manually selecting the features that might contribute to grouping behavior based on the types following their definition presented in Section 2.2.2 (p. 8).
- **Limitations** imposed by the project in terms of Internet connectivity and virtualization. As many malicious types require an active Internet connection to perform their intended task, this behavioral data was not present in this project but may have been present if the malware infected a real-world machine with full Internet access.
- **Behavioral data** collected by running the malware in a secure environment may be different from the data used by AV vendors to label the detected samples. As 18% of malicious software detect that are being executed in a contained environment, see [Wueest, 2014], it can be discussed that their behavior may change, resulting in different information collected from behavioral analysis.

The clusters created by Self Organizing Map contain behavioral data that need to be further analyzed in order to determine what kind of information they contain. Upon further researching the cluster contents, cluster-based classification can then be applied to classify new malware samples. The evaluation of the cluster-based classification should present improved accuracy over classification using AV labels. This can be explained by the fact that the classes were created based on the data and the classification algorithm will have high predictive power as the classes are well separated.

Determined from the long running times, a multi-threaded implementation of SOM might be a better solution to be able to further tweak its parameters, use significantly larger grid sizes and at the same time meet the requirements of analyzing a large number of malware samples in a short period of time. Future work should as well test other implementations of SOM:

- LVQ-SOM - where the closest BMU positively updates its weights, just like the tested SOM, however other nodes will negatively update their weights. These changes will have a significant impact on creation of clusters as the winning node will move closer to the input vector while other node will move away from it, allowing better discrimination between similar groups of behavioral data.
- Batch-SOM - represents another implementation of SOM where all input vectors are presented to the map before the updating step begins and not incrementally as it has been used in this project.

Furthermore it can be discussed that clustering malware behavior represents an important piece in the battle with malicious software. It can be used after detection and before classification as described in Figure 7.9.

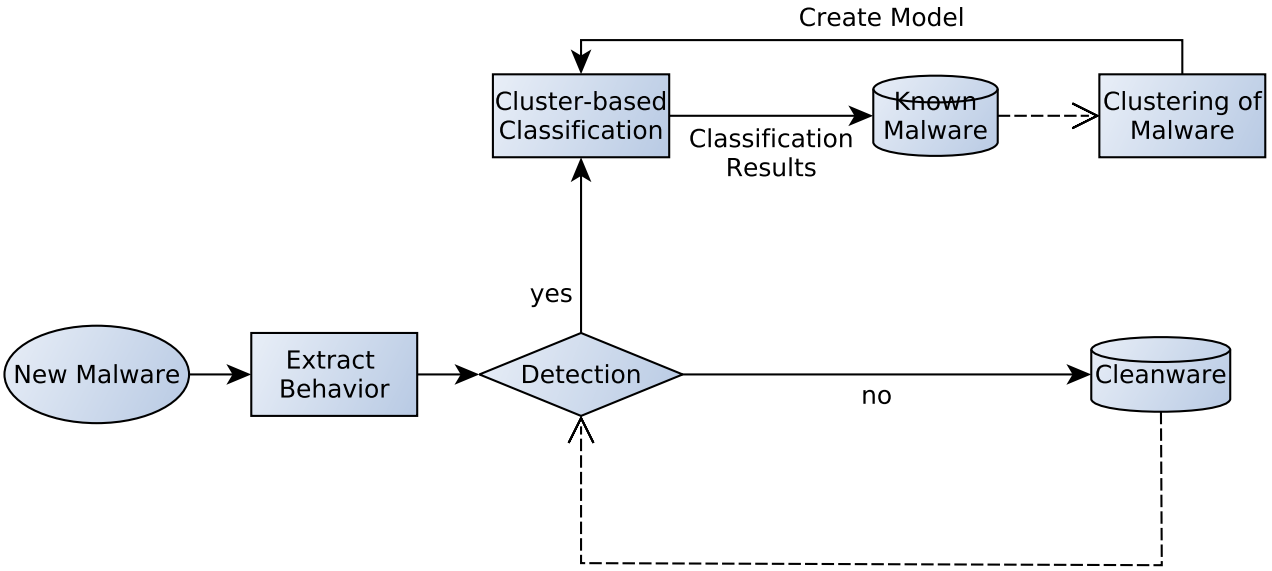


Figure 7.9: Proposed system for future work.

Figure 7.9 can be split in three main parts:

1. **Detection** - is denoted in the diagram as a `yes` or `no` decision. It can be implemented using either a binary classification or as a two cluster unsupervised learning algorithm. For classification a database of cleanware is needed along with their behavioral information. For unsupervised learning, the approach is a bit more difficult as the correct features must be selected in order to make the binary decision between two clusters as accurate as possible.
2. **Clustering** - It is dependent on a database of known malware along with their respective behavioral data in order to cluster and create an input for the classification algorithm. This module was implemented and evaluated in this project.
3. **Classifcation** - Requires the input of the clustering algorithm and its purpose is to use the information provided by the clustering algorithm to evaluate and assigns labels to new malware at a much faster rate.

This section has presented the discussion part of the project where potential improvements for future work have been addressed. Furthermore, a unique detection and labeling system has been proposed that could potentially keep up with the exponential increase of malware infections.

INTENTIONALLY LEFT BLANK

Part V

Conclusion

Chapter 8

Conclusion

In this project, the problem of classification using inconsistent labels has been addressed based on the results gathered from previous work [Pirscoveanu et al., 2015]. Observing the differences in detection and False Positive Rates that differ between AV vendors, the importance of correct labeling when dealing with malicious infections has been researched. While analyzing the already known malware types, it has been noted that different malicious software have a large range of purposes and different infection techniques that may not always be detected using the common static analysis tools. Along with the problem of an increasing amount of malware each day, the need for an automated system that can cope with the increasing number of malware, while at the same time grouping similar malware, has been identified. The advantages of the latter are to prevent infections, or clean an already infected system with the appropriate method without making use of AV generated labels. Having already decided that the malware behavior data will be generated from a dynamic based analysis using Cuckoo and multiple virtualized environments, the following problem statement was formed:

How to cluster malware behavior in order to discover similarities, using Machine Learning applied on behavioral data generated using dynamic analysis?

Having collected behavioral data and detection labels from approximately 270,000 malware samples, AV vendors were evaluated on Completeness, Consistency and Correctness criteria. Based on the very low False Positive Rate and the use of CARO naming convention, Microsoft was selected as the provider of labels for evaluation. Furthermore, a Majority Vote between all vendors using a tokenized Levenshtein ratio was also to evaluate and determine the similarities between groups generated via unsupervised learning. Passed and Failed API calls along with their respective return codes were used to generate the behavioral profile of each malicious sample after which they were represented in matrix form, by their associated frequency. Given that the number of features was fairly large, multiple feature selection methods were analyzed. This analysis resulted in a co-variance based PCA to be chosen. The Gap Statistics method was selected for computing the number of clusters with the Self Organizing Map as preference for clustering the selected features. This provides an innovative way of preserving topological properties of higher dimensional data into a two dimensional elastic grid.

The results were split into optimal number of clusters, feature and algorithm evaluations that have shown interesting results in terms of consistency and created groups. The optimal number of clusters based on the evaluated data-set has shown no similarities with the number of different AV generated types present in the extracted labels. The Combination matrix used for evaluation of the SOM algorithm results, was selected based on the information that the features may hold. By ranging the grid size of the created map, the results were evaluated in terms of similarity of clusters with type labels from the Majority vote, as well as based on the information extracted from the visualization of the map. It has been concluded

that the created groups do not match the type labels presented by the AV vendors, but instead they hold behavioral information that, with future research, may determine important information about the intentions of the malware.

It may be interesting in future studies to examine the use of different feature representation for creating the behavioral profile of the malware, as this could lead to better groups generated by the clusterer. Furthermore, implementing a multi-threaded variant of Self Organizing Map may result in exploring the accuracy of much larger grid sizes that should define a better delimitation between malware behavior groups.

Even though the evaluation has returned negative results in terms of similarities between the behavioral data of the clusters and the extracted labels, an initial proposal of a cluster-based classification system was introduced. This method mainly relies on creating classes from collected behavioral data, which can be used in a classification algorithm that will perform as intended. This provides a starting point for future classification work that will use cluster created labels from dynamically collected data instead of labels provided by the AV vendors.

INTENTIONALLY LEFT BLANK

Part VI

References

Chapter 9

List of references

[Abbas, 2007] Abbas, O. A. (2007). Comparisons between data clustering algorithms.

[Abdi and Williams, 2010] Abdi, H. and Williams, L. J. (2010). Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(4):433–459.

[Ahmad et al., 2015] Ahmad, S., Ahmad, S., Xu, S., and Li, B. (2015). Next generation malware analysis techniques and tools. In *Electronics, Information Technology and Intellectualization: Proceedings of the International Conference EITI 2014, Shenzhen, 16-17 August 2014*, page 17. CRC Press.

[Anubis, 2014] Anubis (2014). Anubis - malware analysis for unknown binaries. `https://anubis.iseclab.org/`.

[Arthur and Vassilvitskii, 2007] Arthur, D. and Vassilvitskii, S. (2007). k-means++: the advantages of carefull seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035.

[AV-Comparatives, 2015] AV-Comparatives (2015). Av-comparatives - independent tests of anti-virus software. `http://www.av-comparatives.org/about-us/`.

[AV-TEST, 2015] AV-TEST (2015). Av-test - antivirus & malware research. `http://www.av-test.org/en/about-the-institute/`.

[Bação et al., 2005] Bação, F., Lobo, V., and Painho, M. (2005). Self-organizing maps as substitutes for k-means clustering. In *Computational Science–ICCS 2005*, pages 476–483. Springer.

[Bayer et al., 2009] Bayer, U., Comparetti, P. M., Hlauschek, C., Kruegel, C., and Kirda, E. (2009). Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer.

[Brumley et al., 2008] Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Song, D., and Yin, H. (2008). Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer.

[BullGuard, 2015] BullGuard (2015). Malware - definition, history and classification. `http://www.bullguard.com/da/bullguard-security-center/pc-security/computer-threats/malware-definition,-history-and-classification.aspx`.

[Chodorow and Dirolf, 2010] Chodorow, K. and Dirolf, M. (2010). *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition.

[Cisco, 2015] Cisco (2015). What is the difference: Viruses, worms, trojans, and bots? Online. `http://www.cisco.com/web/about/security/intelligence/virus-worm-diffs.html`.

[Crockford, 2013] Crockford, D. (2013). The json data interchange format. Technical report, Technical report, ECMA International, October.

[Do and Batzoglou, 2008] Do, C. B. and Batzoglou, S. (2008). What is the expectation maximization algorithm? *Nature biotechnology*, 26(8):897–899.

[Egele et al., 2012] Egele, M., Scholte, T., Kirda, E., and Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6.

[FitzGerald, 2002] FitzGerald, N. (2002). A virus by any other name: Towards the revised caro naming convention. *Proc. AVAR*, pages 141–166.

[Gashi et al., 2013] Gashi, I., Sobesto, B., Mason, S., Stankovic, V., and Cukier, M. (2013). A study of the relationship between antivirus regressions and label changes. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 441–450.

[Gentleman and Carey, 2008] Gentleman, R. and Carey, V. (2008). Unsupervised machine learning. In *Bioconductor Case Studies*, pages 137–157. Springer.

[Gorecki et al., 2011] Gorecki, C., Freiling, F. C., Kührer, M., and Holz, T. (2011). Trumanbox: Improving dynamic malware analysis by emulating the internet. In *Stabilization, Safety, and Security of Distributed Systems*, pages 208–222. Springer.

[Griffin et al., 2009] Griffin, K., Schneider, S., Hu, X., and Chiueh, T.-c. (2009). Automatic generation of string signatures for malware detection. In Kirda, E., Jha, S., and Balzarotti, D., editors, *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 101–120. Springer Berlin Heidelberg.

[Guyon and Elisseeff, 2003] Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182.

[Hamming, 1950] Hamming, R. W. (1950). Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160.

[Hartigan and Wong, 1979] Hartigan, J. A. and Wong, M. A. (1979). Algorithm as 136: A k-means clustering algorithm. *Applied statistics*, pages 100–108.

[Hirschberg, 1997] Hirschberg, D. (1997). Serial computations of levenshtein distances.

[Hou et al., 2015] Hou, S., Chen, L., Tas, E., Demihovskiy, I., and Ye, Y. (2015). Cluster-oriented ensemble classifiers for intelligent malware detection. In *Semantic Computing (ICSC), 2015 IEEE International Conference on*, pages 189–196.

[Jolliffe, 2002] Jolliffe, I. (2002). *Principal component analysis*. Wiley Online Library.

[Jones et al., 01] Jones, E., Oliphant, T., Peterson, P., et al. (2001–). SciPy: Open source scientific tools for Python. `http://www.scipy.org/`.

[Kent, 1983] Kent, J. T. (1983). Information gain and a general measure of correlation. *Biometrika*, 70(1):163–173.

[Kephart and Arnold, 1994] Kephart, J. O. and Arnold, W. C. (1994). Automatic extraction of computer virus signatures. In *4th virus bulletin international conference*, pages 178–184.

[Kohonen, 1989] Kohonen, T. (1989). *Self-organization and Associative Memory: 3rd Edition*. Springer-Verlag New York, Inc., New York, NY, USA.

[Kotsiantis et al., 2007] Kotsiantis, S. B., Zaharakis, I., and Pintelas, P. (2007). Supervised machine learning: A review of classification techniques.

[Microsoft, 2006] Microsoft (2006). Introduction to win32/win64. https://technet.microsoft.com/en-us/library/bb496995.aspx.

[Microsoft, 2008a] Microsoft (2008a). File management functions. https://msdn.microsoft.com/en-us/library/windows/desktop/aa364232%28v=vs.85%29.aspx.

[Microsoft, 2008b] Microsoft (2008b). Mutex functions. https://msdn.microsoft.com/en-us/library/windows/desktop/ms686360(v=vs.85).aspx#mutex_functions.

[Microsoft, 2008c] Microsoft (2008c). Registry functions. https://msdn.microsoft.com/en-us/library/windows/desktop/ms724875(v=vs.85).aspx.

[Microsoft, 2008d] Microsoft (2008d). Registrykey lass. https://msdn.microsoft.com/en-us/library/microsoft.win32.registrykey%28v=vs.110%29.aspx.

[Microsoft, 2015a] Microsoft (2015a). Microsoft developer network. Online. https://msdn.microsoft.com/en-us/.

[Microsoft, 2015b] Microsoft (2015b). User account control. https://msdn.microsoft.com/en-us/library/windows/desktop/bb648649%28v=vs.85%29.aspx.

[Mohaisen and Alrawi, 2014] Mohaisen, A. and Alrawi, O. (2014). Av-meter: An evaluation of antivirus scans and labels. In Dietrich, S., editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 8550 of *Lecture Notes in Computer Science*, pages 112–131. Springer International Publishing.

[Mohaisen et al., 2014] Mohaisen, A., Alrawi, O., Larson, M., and McPherson, D. (2014). Towards a methodical evaluation of antivirus scans and labels. In Kim, Y., Lee, H., and Perrig, A., editors, *Information Security Applications*, volume 8267 of *Lecture Notes in Computer Science*, pages 231–241. Springer International Publishing.

[Moser et al., 2007] Moser, A., Kruegel, C., and Kirda, E. (2007). Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE.

[Motiee et al., 2010] Motiee, S., Hawkey, K., and Beznosov, K. (2010). Do windows users follow the principle of least privilege?: Investigating user account control practices. In *Proceedings of the Sixth Symposium on Usable Privacy and Security*, SOUPS '10, pages 1:1–1:13, New York, NY, USA. ACM.

[NumPy, 2015] NumPy (2015). Numpy. http://www.numpy.org/.

[Pirscoveanu et al., 2015] Pirscoveanu, R. S., Hansen, S. S., Larsen, T. M. T., Czech, A., Stevanovic, M., and Pedersen, J. M. (2015). Analysis of malware behavior: Type classification using machine learning. Attached as Appendix.

[Ratcliff and Metzener, 1988] Ratcliff, J. W. and Metzener, D. E. (1988). Pattern-matching-the gestalt approach. *Dr Dobbs Journal*, 13(7):46.

[Roth and Lange, 2003] Roth, V. and Lange, T. (2003). Feature selection in clustering problems. In *Advances in neural information processing systems*, page None.

[Scikit, 2015] Scikit (2015). Scikit-learn , machine learning in python. http://scikit-learn.org/stable/.

[Sharif et al., 2008] Sharif, M., Yegneswaran, V., Saidi, H., Porras, P., and Lee, W. (2008). Eureka: A framework for enabling static malware analysis. In *Computer security-ESORICS 2008*, pages 481–500. Springer.

[Stefanovič and Kurasova, 2011] Stefanovič, P. and Kurasova, O. (2011). Influence of learning rates and neighboring functions on self-organizing maps. In Laaksonen, J. and Honkela, T., editors, *Advances in Self-Organizing Maps*, volume 6731 of *Lecture Notes in Computer Science*, pages 141–150. Springer Berlin Heidelberg.

[Symantec, 2009] Symantec (2009). What are malware, viruses, spyware, and cookies, and what differentiates them ? Online. http://www.symantec.com/connect/articles/what-are-malware-viruses-spyware-and-cookies-and-what-differentiates-them.

[Symantec, 2014] Symantec (2014). Trojan.zbot. Online. http://www.symantec.com/security_response/writeup.jsp?docid=2010-011016-3514-99.

[Thorndike, 1953] Thorndike, R. (1953). Who belongs in the family? *Psychometrika*, 18(4):267–276.

[Tibshirani et al., 2001] Tibshirani, R., Walther, G., and Hastie, T. (2001). Estimating the number of clusters in a data set via the gap statistic. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(2):411–423.

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

[Vejdemo-Johansson, 2013] Vejdemo-Johansson, M. (2013). A python implementation of the gap statistic. https://gist.github.com/michiexile/5635273.

[VirusShare, 2014] VirusShare (2014). Virusshare malware database. Online. http://virusshare.com/.

[VirusTotal, 2015] VirusTotal (2015). Virustotal - free online virus, malware and url scanner. Online. https://www.virustotal.com/.

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

[Wikipedia, 2005a] Wikipedia (2005a). Mutual exclusion. http://en.wikipedia.org/wiki/Mutual_exclusion.

[Wikipedia, 2005b] Wikipedia (2005b). Windows registry. http://en.wikipedia.org/wiki/Windows_Registry.

[Wikipedia, 2008] Wikipedia (2008). Object linking and embedding. http://en.wikipedia.org/wiki/Object_Linking_and_Embedding.

[Wikipedia, 2015a] Wikipedia (2015a). Antivirus software. http://en.wikipedia.org/wiki/Antivirus_software.

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

[Wikipedia, 2015c] Wikipedia (2015c). Windows api. http://en.wikipedia.org/wiki/Windows_API.

[Wueest, 2014] Wueest, C. (2014). Threats to virtual environments. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/threats_to_virtual_environments.pdf.

Part VII

Appendices

Appendix A

Detection Rate Code

Code snippet A.1: Extracting AV results to PKL serialized file

```
def getDetectionRate(db, limit=2):

    pushLog("Starting Extraction of AV labels. Limit %s" % limit)
    customLog("Starting Extraction of AV labels. Limit %s" % limit)

    start_time = datetime.now()
    temp = db.analysis.find_one()

    detected = list()
    labels = list()
    samples = list()


    listOfAV = list(temp['virustotal']['scans'].keys())
    leng = len(listOfAV)

    AVquery = db.analysis.find().limit(limit)

    pbar = initProgress("Extracting AV: ", AVquery.count(True))
    d = 0

    for item in AVquery:

        tmpDetected = [False] * leng
        tmpLabels = [""] * leng

        for ii in range(0, len(listOfAV)):
            try:
              if item['virustotal']['scans'][listOfAV[ii]]['detected'] == True :
                tmpDetected[ii] = True
                tmpLabels[ii] = str(item['virustotal']['scans'][listOfAV[ii]]['result'])
              #tmpDetected[ii] = item['virustotal']['scans'][listOfAV[ii]]['detected']
              #tmpLabels[ii] = item['virustotal']['scans'][listOfAV[ii]]['result']
            except:
              tmpDetected[ii] = False
              tmpLabels[ii] = ""
        sample = str(item['target']['file']['name'])

        detected.append(tmpDetected)
        labels.append(tmpLabels)
        samples.append(sample)

```

```
43          #print detected
44          #print labels
45          #print samples
46
47          pbar.update(d+1)
48          d += 1
49
50      end_time = datetime.now()
51      pushLog("Finished Extracting AV labels. " + str(AVquery.count(True)) + " done in " + '{}
            '.format(end_time - start_time))
52      customLog("Finished Extracting AV labels. " + str(AVquery.count(True)) + " done in " + '
            {}'.format(end_time - start_time))
53
54      try:
55
56        AVMat = dict()
57        AVMat['samples'] = samples
58        AVMat['detected'] = detected
59        AVMat['labels'] = labels
60        AVMat['AV'] = listOfAV
61
62        customLog('Saving AV results to \'Representations/AVDetection.pkl\'')
63        with open('AVDetection.pkl', 'wb') as handle:
64          pickle.dump(AVMat, handle)
65
66        #mat4py.savemat('Representations/AVDetection.mat', AVMat)
67        print 'Saved'
68
69        pushLog("Saved data to PKL file")
70      except Exception as e:
71        pushLog("Failed to save data to PKL file." +  str(e))
72        print e
```

Appendix B

ARFF Exporter

Code snippet B.1: ARFF export with and without nominal classes.

```
def dumpArff(filez, dataz,relation,names):
    f = open(filez, 'wb')
    data = dict()
    data['data'] = dataz
    attrList = [0] * len(names)
    for x in range(0, len(names)):
        attrList[x] = (names[x] , u'NUMERIC')
    data['attributes'] = attrList
    data['relation'] = relation
    arff.dump(data, f)
    f.close()

def dumpArffNominal(filez,dataz,relation,names,nom):
    f = open(filez, 'wb')
    data = dict()
    data['data'] = dataz
    attrList = [(u'class' , list(nom))]
    print attrList
    data['attributes'] = attrList
    data['relation'] = relation
    arff.dump(data, f)
    f.close()
```

Appendix C

Complete API List

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 D

Passed API List

API Name	Occurences	API Name	Occurences	API Name	Occurences
LdrGetProcedureAddress	100,617,968	**ExitThread**	635,374	**CreateRemoteThread**	29,138
GetSystemMetrics	86,133,172	**DeleteFileA**	613,862	**NtSuspendThread**	28,405
NtQueryKey	54,509,185	**NtSetValueKey**	610,362	**URLDownloadToFileW**	26,986
NtQueryValueKey	47,218,656	**WriteProcessMemory**	561,311	**recvfrom**	25,896
RegCloseKey	47,110,126	**NtResumeThread**	559,937	**InternetConnectA**	24,074
NtOpenKeyEx	44,528,796	**ReadProcessMemory**	527,199	**HttpOpenRequestA**	22,799
RegOpenKeyExW	27,219,780	**socket**	521,811	**NtWriteVirtualMemory**	22,735
NtDelayExecution	23,578,113	**NtOpenDirectoryObject**	497,575	**ShellExecuteExW**	19,593
NtSetInformationFile	21,406,873	**CreateProcessInternalW**	475,746	**shutdown**	18,724
NtReadFile	20,645,451	**recv**	459,896	**InternetConnectW**	18,682
NtCreateFile	18,658,372	**closesocket**	444,933	**HttpOpenRequestW**	18,651
LdrLoadDll	17,898,802	**anomaly**	440,898	**sendto**	17,672
NtQueryInformationFile	17,308,131	**VirtualProtectEx**	431,079	**CreateServiceA**	17,003
NtReadVirtualMemory	16,451,461	**ExitProcess**	430,560	**StartServiceA**	14,677
RegOpenKeyExA	15,473,157	**setsockopt**	377,506	**InternetReadFile**	11,569
RegQueryValueExA	15,148,061	**ioctlsocket**	313,836	**MoveFileWithProgressW**	11,257
NtEnumerateKey	14,242,601	**gethostbyname**	310,106	**WSASend**	11,049
ZwMapViewOfSection	13,821,942	**OpenSCManagerA**	278,517	**CopyFileW**	9,935
FindFirstFileExW	12,355,900	**WSAStartup**	275,969	**RemoveDirectoryA**	9,292
NtOpenKey	11,409,990	**OpenSCManagerW**	262,831	**listen**	6,292
NtWriteFile	10,400,296	**OpenServiceW**	247,610	**RegDeleteValueA**	5,132
NtOpenFile	9,909,508	**send**	247,146	**NtSaveKey**	4,888
LdrGetDllHandle	9,203,419	**FindWindowA**	219,741	**ControlService**	4,693
RegEnumValueW	9,066,952	**OpenServiceA**	214,051	**RemoveDirectoryW**	4,299
RegQueryValueExW	8,000,332	**NtOpenThread**	210,536	**NtTerminateThread**	3,037
NtCreateSection	6,664,476	**RegQueryInfoKeyA**	207,957	**RegDeleteKeyW**	2,960
NtQueryDirectoryFile	6,563,411	**SetWindowsHookExW**	207,361	**NtTerminateProcess**	2,897
GetCursorPos	6,332,962	**FindWindowExW**	198,590	**WSASocketA**	1,966
NtDeviceIoControlFile	6,317,501	**getaddrinfo**	162,474	**CopyFileExW**	1,663
NtFreeVirtualMemory	5,801,849	**FindWindowW**	162,058	**StartServiceW**	1,658
NtProtectVirtualMemory	4,114,030	**NtCreateThreadEx**	160,692	**system**	1,638
RegSetValueExA	3,980,099	**InternetCloseHandle**	143,024	**DnsQuery_A**	1,030
RegCreateKeyExW	3,511,034	**bind**	134,005	**DeleteService**	969
RegEnumKeyW	2,946,500	**CreateDirectoryW**	123,384	**HttpSendRequestW**	400
RegCreateKeyExA	2,780,508	**CopyFileA**	108,360	**CreateDirectoryExW**	388
NtCreateKey	2,179,170	**LookupPrivilegeValueW**	107,724	**CreateServiceW**	293
NtOpenSection	2,013,254	**DeleteFileW**	95,744	**FindFirstFileExA**	282
RegSetValueExW	1,857,952	**SetWindowsHookExA**	95,090	**accept**	254
RegEnumKeyExA	1,824,804	**RegDeleteValueW**	94,810	**NtDeleteValueKey**	218
WriteConsoleW	1,817,077	**InternetOpenA**	70,101	**RtlCreateUserThread**	205
RegEnumKeyExW	1,562,106	**connect**	63,726	**InternetWriteFile**	148
RegEnumValueA	1,548,660	**WSARecv**	54,148	**WSARecvFrom**	84
RegQueryInfoKeyW	1,479,023	**InternetOpenW**	53,563	**NtLoadKey**	75
NtCreateMutant	1,476,677	**NtDeleteKey**	51,626	**WSASendTo**	59
DeviceIoControl	1,456,921	**NtGetContextThread**	48,880	**NtMakeTemporaryObject**	54
FindWindowExA	1,291,115	**GetAddrInfoW**	46,740	**NtQueryMultipleValueKey**	29
NtOpenMutant	1,273,191	**UnhookWindowsHookEx**	42,886	**NtSaveKeyEx**	10
CreateThread	1,247,037	**RegDeleteKeyA**	40,796	**NtDeleteFile**	9
NtEnumerateValueKey	987,772	**NtSetContextThread**	38,039	**NtCreateUserProcess**	5
select	920,992	**NtCreateNamedPipeFile**	33,553	**InternetOpenUrlA**	4
WriteConsoleA	665,686	**WSASocketW**	30,961	**NtCreateProcess**	2

Appendix E

Failed API List

API Name	Occurences	API Name	Occurences	API Name	Occurences
NtOpenKeyEx	79,275,174	**accept**	47,377	**HttpOpenRequestA**	688
NtQueryValueKey	65,124,562	**InternetOpenUrlA**	47,106	**NtReadVirtualMemory**	675
RegOpenKeyExW	21,814,148	**recv**	46,301	**DeleteService**	526
RegQueryValueExW	16,424,917	**NtSetInformationFile**	38,526	**RegSetValueExW**	428
RegQueryValueExA	15,099,798	**NtEnumerateValueKey**	37,231	**WSARecvFrom**	350
FindWindowA	13,104,421	**getaddrinfo**	35,896	**CopyFileExW**	314
NtOpenMutant	9,857,883	**InternetReadFile**	32,364	**NtSetContextThread**	265
NtQueryInformationFile	7,717,039	**WSARecv**	30,710	**NtSetValueKey**	242
NtDeviceIoControlFile	7,353,980	**RegSetValueExA**	24,240	**RegCreateKeyExW**	221
RegCloseKey	6,410,365	**NtGetContextThread**	24,096	**RegQueryInfoKeyW**	198
NtOpenFile	5,083,877	**HttpSendRequestA**	23,238	**LookupPrivilegeValueW**	145
FindWindowExA	3,157,206	**select**	22,009	**NtDeleteValueKey**	120
CreateDirectoryW	2,776,225	**NtSuspendThread**	21,828	**DnsQuery_A**	99
RegOpenKeyExA	2,704,966	**NtCreateKey**	21,000	**sendto**	68
NtCreateFile	2,488,322	**InternetOpenUrlW**	20,619	**GetAddrInfoW**	63
NtQueryDirectoryFile	2,342,135	**WriteProcessMemory**	19,081	**InternetOpenA**	44
CreateProcessInternalW	2,041,965	**send**	18,946	**InternetConnectW**	41
NtOpenKey	1,967,197	**closesocket**	18,250	**NtWriteVirtualMemory**	35
FindWindowW	1,750,707	**HttpSendRequestW**	18,184	**WSAStartup**	23
LdrGetDllHandle	1,746,503	**ShellExecuteExW**	16,309	**socket**	16
NtQueryKey	1,677,922	**bind**	13,785	**NtQueryMultipleValueKey**	14
NtEnumerateKey	1,237,287	**NtCreateSection**	13,086	**CreateServiceW**	13
RegDeleteValueW	1,136,362	**InternetCloseHandle**	12,049	**CreateDirectoryExW**	7
NtOpenSection	1,051,046	**recvfrom**	11,320	**OpenSCManagerW**	7
RegEnumValueA	943,180	**StartServiceA**	9,723	**NtSaveKeyEx**	6
LdrGetProcedureAddress	888,038	**CreateRemoteThread**	7,842	**NtMakeTemporaryObject**	3
DeleteFileA	735,635	**RegDeleteKeyW**	6,938	**TransmitFile**	3
NtReadFile	648,295	**OpenServiceW**	6,206	**NtLoadKey**	2
NtProtectVirtualMemory	585,831	**RegCreateKeyExA**	6,034	**NtSaveKey**	2
LdrLoadDll	525,667	**listen**	5,759	**OpenSCManagerA**	2
RegEnumKeyExA	498,939	**VirtualProtectEx**	3,527	**WSASend**	1
RegDeleteValueA	427,499	**NtResumeThread**	3,327	**NtCreateNamedPipeFile**	1
RegEnumValueW	427,444	**MoveFileWithProgressW**	3,073	**NtDeleteFile**	1
connect	419,036	**setsockopt**	2,921	**SetWindowsHookExW**	1
RegEnumKeyW	388,458	**CopyFileW**	2,877	**InternetOpenW**	1
DeleteFileW	384,353	**CreateThread**	2,768		
RegDeleteKeyA	217,093	**RegQueryInfoKeyA**	2,455		
CopyFileA	206,269	**NtCreateMutant**	2,413		
DeviceIoControl	177,781	**CreateServiceA**	1,898		
FindWindowExW	168,507	**SetWindowsHookExA**	1,888		
ReadProcessMemory	131,149	**gethostbyname**	1,883		
RemoveDirectoryA	120,806	**NtTerminateProcess**	1,736		
RegEnumKeyExW	114,209	**ioctlsocket**	1,298		
ZwMapViewOfSection	101,130	**NtTerminateThread**	1,247		
NtWriteFile	79,035	**NtCreateThreadEx**	1,202		
UnhookWindowsHookEx	68,093	**RemoveDirectoryW**	1,141		
URLDownloadToFileW	62,144	**NtOpenThread**	1,032		
NtFreeVirtualMemory	61,529	**InternetConnectA**	891		
ControlService	58,517	**StartServiceW**	794		
OpenServiceA	54,948	**ExitWindowsEx**	783		
shutdown	54,722	**NtDeleteKey**	690		

Appendix F

Failed API Return Codes List

Return Code	Occurences	Return Code	Occurences
STATUS_OBJECT_NAME_NOT_FOUND	159,645,148	**STATUS_BAD_NETWORK_NAME**	1,018
STATUS_WAIT_2	57,588,792	**STATUS_INVALID_PARAMETER_3**	1,013
STATUS_WAIT_0	25,172,618	**STATUS_INVALID_PARAMETER_2**	1,010
STATUS_OBJECT_TYPE_MISMATCH	7,677,093	**STATUS_INVALID_CID**	960
0x00000006	7,041,718	**0x800c0005**	928
STATUS_INVALID_PARAMETER	3,330,110	**STATUS_PIPE_BROKEN**	807
STATUS_FILE_IS_A_DIRECTORY	2,912,852	**STATUS_THREAD_IS_TERMINATING**	777
STATUS_NOT_FOUND	2,741,337	**STATUS_OBJECT_PATH_SYNTAX_BAD**	614
STATUS_BUFFER_OVERFLOW	2,644,070	**0x800c0004**	607
STATUS_DLL_NOT_FOUND	2,253,822	**0x800c0007**	559
STATUS_NO_MORE_FILES	2,234,140	**STATUS_UNSUCCESSFUL**	551
STATUS_PENDING	1,852,968	**0x00002afc**	534
STATUS_BUFFER_TOO_SMALL	1,678,133	**STATUS_PARTIAL_COPY**	509
STATUS_NO_MORE_ENTRIES	1,274,075	**0xc0010019**	495
0xffffffff	692,869	**STATUS_CANNOT_DELETE**	490
STATUS_END_OF_FILE	561,050	**STATUS_INVALID_USER_BUFFER**	473
STATUS_INVALID_PAGE_PROTECTION	559,705	**STATUS_DATATYPE_MISALIGNMENT**	444
STATUS_ENTRYPOINT_NOT_FOUND	541,572	**STATUS_INVALID_DEVICE_REQUEST**	431
STATUS_OBJECT_PATH_NOT_FOUND	464,263	**STATUS_NOT_IMPLEMENTED**	429
STATUS_INVALID_HANDLE	378,056	**STATUS_INVALID_FILE_FOR_SECTION**	357
STATUS_ACCESS_DENIED	312,494	**STATUS_MAPPED_FILE_SIZE_ZERO**	313
STATUS_OBJECT_NAME_COLLISION	221,508	**STATUS_OBJECT_PATH_INVALID**	228
STATUS_OBJECT_NAME_INVALID	149,403	**STATUS_SECTION_PROTECTION**	192
0x00000005	122,219	**STATUS_INVALID_INFO_CLASS**	185
STATUS_NO_SUCH_FILE	108,003	**STATUS_UNABLE_TO_FREE_VM**	167
0x800c0008	54,125	**STATUS_INVALID_IMAGE_WIN_64**	147
STATUS_ACCESS_VIOLATION	49,965	**STATUS_BREAKPOINT**	132
STATUS_PROCEDURE_NOT_FOUND	45,967	**STATUS_ACCOUNT_RESTRICTION**	113
STATUS_SHARING_VIOLATION	39,084	**0x800c0006**	113
STATUS_INVALID_PARAMETER_4	36,703	**STATUS_BAD_NETWORK_PATH**	97
0x00002af9	35,407	**0x80070057**	91
0x000000ea	24,349	**0x00002558**	89
STATUS_NO_MEDIA_IN_DEVICE	24,232	**STATUS_NETWORK_UNREACHABLE**	87
STATUS_MEMORY_NOT_ALLOCATED	17,249	**0x80004005**	74
0x00000057	15,605	**STATUS_PIPE_CLOSING**	67
STATUS_NOT_A_DIRECTORY	15,256	**STATUS_IN_PAGE_ERROR**	58
STATUS_INVALID_IMAGE_NOT_MZ	12,229	**STATUS_INVALID_IMAGE_WIN_16**	57
STATUS_FREE_VM_NOT_AT_BASE	9,870	**0x000003f2**	49
STATUS_PIPE_EMPTY	6,604	**STATUS_INVALID_CONNECTION**	46
0x800c000d	5,633	**0xc0010006**	45
STATUS_PIPE_NOT_AVAILABLE	5,506	**STATUS_SECTION_TOO_BIG**	37
STATUS_DEVICE_NOT_READY	3,265	**STATUS_PRIVILEGED_INSTRUCTION**	31
STATUS_INVALID_ADDRESS_COMPONENT	2,853	**STATUS_INVALID_SYSTEM_SERVICE**	29
STATUS_UNABLE_TO_DELETE_SECTION	2,592	**STATUS_SINGLE_STEP**	24
STATUS_DLL_INIT_FAILED	2,493	**0x0000276c**	23
0x000000a1	2,149	**STATUS_PRIVILEGE_NOT_HELD**	23
STATUS_PROCESS_IS_TERMINATING	1,730	**STATUS_ILLEGAL_INSTRUCTION**	22
STATUS_DELETE_PENDING	1,690	**STATUS_DISK_FULL**	20
STATUS_INVALID_IMAGE_FORMAT	1,637	**STATUS_DUPLICATE_OBJECTID**	19
STATUS_ORDINAL_NOT_FOUND	1,593	**STATUS_SXS_ASSEMBLY_NOT_FOUND**	19
STATUS_CONFLICTING_ADDRESSES	1,222	**0x00002afb**	18

Appendix G

SOM Family Evaluation

Labels \ Cluster	1	2	3	4
Hupigon	146	406	115	7529
Zlob	98	292	105	4964
Small	121	348	83	4646
IframeRef	2	2412	1	2541

Table G.1: Evaluation matrix for 2x2 SOM grid on Microsoft Families.

Labels \ Cluster	1	2	3	4	5	6
Hupigon	65	427	22	7592	42	48
Zlob	45	298	20	5011	43	42
Small	47	373	12	4700	25	41
IframeRef	3	2070	0	2883	0	0
Frethog	35	337	22	4453	37	34
OnLineGames	42	285	7	4068	25	29

Table G.2: Evaluation matrix for 2x3 SOM grid on Microsoft Families.

Appendix H

SOM 4x4 Type Evaluation

Labels \ Cluster	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Trojan	381	21384	83	307	1406	765	233	520	441	126	276	389	97210	518	236	284
Backdoor	1	462	1	28	239	26	18	56	132	4	18	140	17542	23	17	9
Adware	0	2237	0	0	19	0	0	34	1	0	5	15	5889	14	35	4
Malware	0	150	0	0	20	1	0	10	2	2	2	3	1976	1	4	1
Worm	0	146	2	2	23	5	1	9	10	13	0	9	1390	5	2	3
Virus	0	297	0	0	1	0	0	2	1	0	0	0	343	0	1	0
Rootkit	0	5	0	0	0	0	0	1	1	0	0	3	170	0	1	0
Spyware	0	2	0	0	0	0	0	0	0	0	0	0	5	0	0	0

Table H.1: Evaluation matrix for 4x4 SOM grid on Majority Labels.

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.

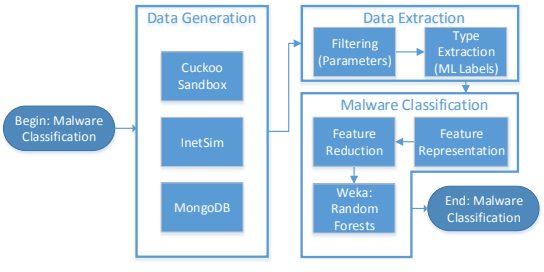


Fig. 1. Overall system flow.

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.

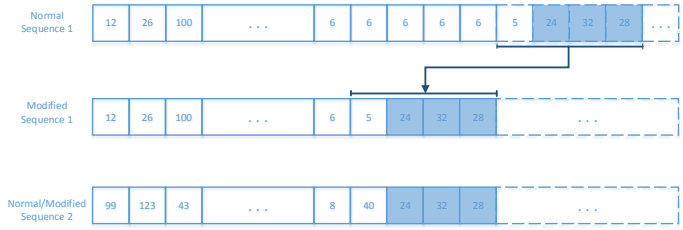


Fig. 2. Sequence Modification.

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.

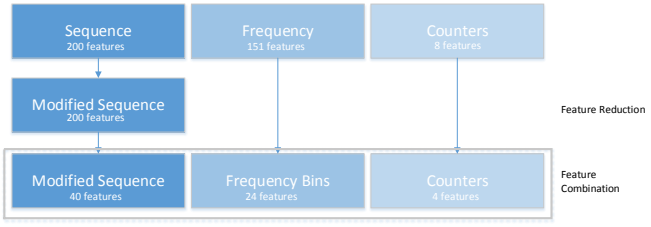


Fig. 3. Construction of the combination matrix.

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.

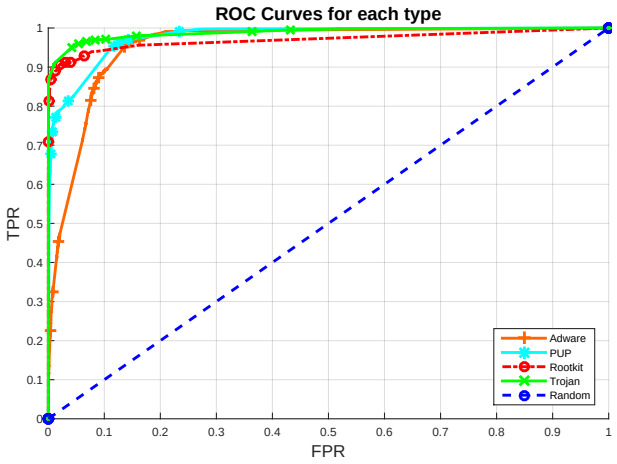


Fig. 4. Receiver Operating Characteristics for the classified types.

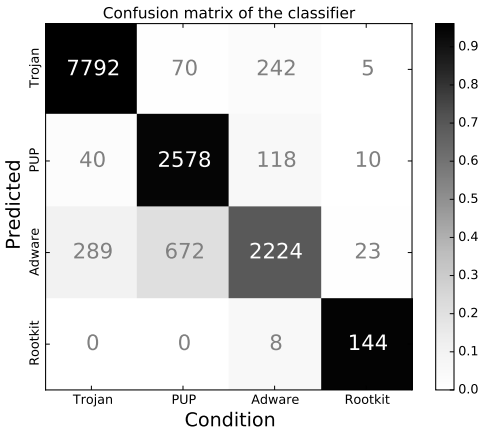


Fig. 5. Results of the classifier in the form of a confusion matrix.

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.