Final thesis

Automatic behavioural analysis of malware

by

Tiziano Santoro

LITH-IDA/ERASMUS-A--10/002--SE

2010-10-22
Automatic behavioural analysis of malware

Tiziano Santoro
University of Rome “La Sapienza”
Linköping Universitet

October 22, 2010
Abstract

With malware becoming more and more diffused and at the same time more sophisticated in its attack techniques, countermeasures need to be set up so that new kinds of threats can be identified and dismantled in the shortest possible time, before they cause harm to the system under attack. With new behaviour patterns like the one shown by polymorphic and metamorphic viruses, static analysis is not any more a reliable way to detect those threats, and behaviour analysis seems a good candidate to fight against the next-generation families of viruses. In this project, we describe a methodology to analyze and categorize binaries solely on the basis of their behaviour, in terms of their interaction with the Operating System, other processes and network. The approach can strengthen host-based intrusion detection systems by a timely classification of unknown but similar malware code. It has been evaluated on a dataset from the research community and tried on a smaller data set from local companies collected at University of Mondragone.
Contents

1 Introduction 3
  1.1 Background ........................................... 3
  1.2 Goal .................................................. 4
  1.3 Motivation ........................................... 4
  1.4 Limitations .......................................... 5
  1.5 Thesis overview ..................................... 6

2 Background 7
  2.1 Malware terminology .................................. 7
  2.2 Malware: a brief history ............................... 9
  2.3 Infection strategies .................................... 9
    2.3.1 Nonresident viruses ................................ 9
    2.3.2 Resident viruses .................................. 10
  2.4 Polymorphic vs. metamorphic code ..................... 10
    2.4.1 Polymorphic code .................................. 10
    2.4.2 Metamorphic code .................................. 11
  2.5 Sandbox environment .................................. 13
    2.5.1 XML reports structure ............................. 13
  2.6 WEKA machine learning software ....................... 14
    2.6.1 ARFF data files .................................... 14
  2.7 Machine learning algorithm ............................ 15
    2.7.1 Introduction to machine learning .................. 15
    2.7.2 Instance-based learning ............................ 15
    2.7.3 Nearest neighbour .................................. 16
    2.7.4 Nearest neighbour with generalization[3] ........... 16
    2.7.5 Distance estimation ................................. 17
    2.7.6 Output model ...................................... 19

3 Automatic analysis approach 20
  3.1 Overview ............................................. 20
  3.2 Feature selection ...................................... 20
  3.3 Feature vector translation .............................. 26
## CONTENTS

4 Testing and evaluation 28
   4.1 Data Set ................................................. 28
   4.2 Evaluation .............................................. 28
   4.3 Results ................................................ 30
      4.3.1 Overall statistics ............................... 30
      4.3.2 Class-specific statistics ...................... 30
      4.3.3 Confusion matrix ............................... 31

5 Discussion 34
   5.1 Conclusions ............................................ 34
   5.2 Future work .......................................... 34

Bibliography 36
Chapter 1

Introduction

This thesis is based on a study performed as a partial fulfillment of a Bachelor Degree in Computer Engineering at the University of Rome “La Sapienza”. The work has been carried out at the Department of Computer and Information Science at Linköping University.

1.1 Background

Almost every person who uses the computer for working, studying, or just entertainment, knows how annoying it is not to be able to properly use it because of viruses, worms, or other annoying situations that can happen when one does not pay enough attention to properly secure his or her machine. Even worse is the case in which one finds oneself in the situation of not being able to access own data, because a virus has compromised all (or most) of the files present on the hard drive, resulting sometimes in loss of time and money. This is inevitable when one blindly relies on the storage capabilities provided by computers, without considering proactive measures such as installing firewalls and anti-virus systems, and planning regular backups of data. Although those systems are at least a step toward a more secure environment, nonetheless a computer is far from being completely secured from any threat at any time just by means of this, since attackers are always looking for new ways to gain unauthorized access to user machines, by developing new and innovative infection strategy that often results in widespread virus or worm infections. Sometimes they are so well-designed that they manage to infect a very large number of machines, and it is not uncommon to hear news about the inconvenience caused by new infections that have not yet been analyzed and for which a solution has not been created.

Although the first viruses ever created were very simple programs, mostly created as jokes between colleagues, the situation has changed with time, especially when virus designers realized that they could make money with their work. An example of this practice are programs that steal personal data, and give the attacker access to the personal accounts of people that use online banking systems, obtaining economic value.

In the history of malware, attack and defense strategies have more or less been
CHAPTER 1. INTRODUCTION

balancing one another like in a repartee, with each of the sides promptly responding to the move of the other.

In this context, companies started to develop anti-malware systems – either software-based, such as anti-virus and malware removal tools, or with added hardware, such as intrusion detection systems or firewalls to be installed over networks. Although at the beginning they focused on a small set of threats that were pretty easy to identify and dismantle, now this has become something more challenging. For example it is not uncommon for anti-malware products to often update themselves (even several times per day) to constantly receive information about newly discovered types of attacks.

The key word here is fast response time, in order to minimize disruptions that can possibly be caused by rapidly spreading infections, especially if related to worms that can use the Internet to spread. Anti-malware software vendors often use so-called “honeynets” as easy targets for the malware to attack them, so that when a new infection is detected, they can quickly generate an identification pattern for it, and distribute the remedy to all their subscribers.

Since honeynets are made up by intentionally unsecured machines, they are targeted by tons of infections every day, and a lot of work has to be done to distinguish brand new infections from the ones that are already known to exist. Moreover, with classic analysis systems it is likely that, even when a small modification is made to an executable malware (for example because its author added a new functionality, or corrected a bug), the existing anti-malware solutions will not be able to identify this new sample as a variation of an already known class of malware, and this would require the intervention of a human actor to manually analyze the new sample and eventually assign it either to an existing or to a new category.

In this context we present a methodology to partially automate this workflow by analysing new samples and assigning them to existing malware classes.

1.2 Goal

The goal of this thesis work is to design and implement an efficient automatic system for the analysis of malware samples collected “in the wild”, allowing a quick categorization of them within pre-existing classes of malware whose behaviour is already known. This should apply in cases in which it is not possible to proceed according to classic static analysis techniques, e.g. when malware mutations or code encryption are in act.

The proposed solution should, given a sufficiently large training set with a number of samples for each known class of malware, be able to determine for a new, untested example – which may not belong to the original data set used for training – what is the class that shows the most similar behaviour to the one shown by the example.

1.3 Motivation

While there currently exist a lot of techniques for static analysis of executable files, most of which are based on signatures and hash calculation, viruses are evolving to gradually
evade the protection offered by this kind of barriers. Nowadays, many viruses take advantage of advanced obfuscation techniques and polymorphic or metamorphic code, in order to avoid the detection by anti-malware products for as long as possible, in the desperate effort to protect themselves and to spread the infection to as many machines as they can. With polymorphic and metamorphic code, static analysis is useless in most of the cases, because current advanced malware can automatically modify its code. In particular, specific binary strings that identify a particular instance of the executables are completely lost every time the virus duplicates itself, either on the same system or on another machine. Most of the metamorphic viruses alter the form of their code while maintaining the very same functionality in terms of high-level interaction with the OS (file system, network, registry, etc.). This leads to the idea of analyzing the behaviour of the executables rather than just looking at their form. One way to do this is to run the executable in a sandbox or in a virtual machine, take a snapshot of the system before and after its execution, and then comparing the two for differences, like created files, registry entries, and so on. This way we can acquire a “behavioural signature” of each malware which does not depend on the particular way its code is written, but only on its meaning and its interactions with the system – in other words, its semantics. This is the approach adopted in the work performed for this thesis.

1.4 Limitations

The work was limited to one approach for automatic analysis due to the limitations in time. The approach was selected in collaboration with the cooperation partners at Real-time Systems Laboratory of Linköping University and at the University of Mondragone, and based on the approach published by Rieck et al. at the University of Mannheim[1].

The approach is additionally limited to the analysis of executables that can run on the Microsoft Windows operating system, due to the sandbox environment that has been used, but it can be easily adapted to the analysis of other types of executables, by using a different sandbox, and by slightly adapting the categories of the feature analyzed in the behaviour of the modules, if needed.

Moreover, the solution as it is implemented now does not allow to distinguish between malicious and innocuous software, in that it will just output the class to which the behaviour looks most similar to, even if this behaviour is actually very different from the analyzed one. This is also the reason for the lack of another important feature that would surely be useful in practice, that is the ability for the system to automatically determine new classes of malware even when the labeling is not known a priori on a sample set of them, thus implementing in practice a clustering algorithm on top of the already present classification algorithm. Both these problems could be at least partially overcome if it were possible to obtain the value of the distance of a sample to its nearest neighbor, so that a threshold could be set on this distance (determined either experimentally for each class during the training phase, or set once and for all at the beginning) so that if a sample is too different from any other existing class, no class will be assigned to it. In this case, the “blank” samples could then be stored in a special set until a significative
number of them showed similar characteristics, and then they would be grouped in a new class, representing a particular behaviour shown by the exemplars that belong to it; still, in this case it is needed the presence of a human actor to determine if the class is representing a malware or just a regular program, because the system would not be able to recognise it in any case.

Another important limitation is that the system is not able to analyze the behaviour of a given module in real-time, while it is running on a user machine, rather it requires an off-line analysis via a sandbox tool which usually consumes a time of some dozens of seconds. This makes it impossible to use the system as a personal protection system, like the ones commonly known as antivirus products. Nevertheless, the approach is to some extent modular, so that if the sandbox module were changed into an on-line monitor system, and as long as the output of such system were in some way compatible with the rest of the system, the same analysis could be performed in real time on running modules, so to detect threats while they are executed.

1.5 Thesis overview

The work is divided in the following way:

Chapter 2 describes the terminology used in the thesis and the tools that were needed to apply the above approach.

Chapter 3 describes the approach that has been proposed as a solution for the automatic classification problem.

Chapter 4 presents the testing methodology used to evaluate the approach, together with some statistical information about the outcome of the tests.

Chapter 5 contains some final notes about the approach, plus some indications of how the work could be improved.
Chapter 2

Background

2.1 Malware terminology

In this section we present a short description of some of the fundamental terminology and concepts related to malware infections, in order to have a better understanding of their inner working.

**Malware** is short for malicious software. It represents the category of programs designed to infiltrate a computer system without the owner’s informed consent. The term is used to mean a variety of forms of hostile, intrusive, or annoying software or program code. Software is considered malware based on the perceived intent of the creator rather than any particular features, and moreover, software can exist which actually performs (or pretends to perform) some kind of useful task for the user some of the time, while switching to a subtle malicious behaviour given certain conditions, such as being run in a particular time and date, on a specific version of the operating system, or with certain services running on the system, and so on. This makes the distinction between “good” and “evil” programs even more fuzzy and subtle, and therein lie the limits of most of the existing anti-malware solution of today.

**Computer virus** is a computer program that has the ability to reproduce itself, with the goal of spreading on as many machines as possible. Usually a virus will make copy of itself either *verbatim* or by applying modification to its own code and structure, but still keeping the same semantic content. A virus needs to be phisically carried onto the target machine in order to infect it, and ways in which this can be done include (but are not limited to) USB drives, CDs, DVDs, floppies, and also the network, in form of email attachments or download links. Anyway, there is the need of a human actor in the spreading procedure, which will either execute (unwillingly) the malicious executable on the target machine, or at least connect the device on which the virus is stored, so that it can then be run by means for example of autorun features present in the operating system.
**Computer worm** is a self-replicating Malware computer program. Differently from regular viruses, worms do not need the action of a human to spread; instead, they often rely on exploiting either known or 0-day (i.e. not yet published) vulnerabilities on the target machine. Common software that can present such vulnerabilities is for example the operating system itself, network services, daemons, and more generally all the software that is reachable from the outside of a machine via a network connection. Usually worms are considered *volatile* in the sense that they execute within the address space of the target service or program, and so they would be terminated when their host program is closed; anyway they can then initiate their own thread or process and also install themselves like regular viruses, so that the infection is not stopped just by a restart of the target service or machine. Worms almost always cause some sort of harm to the network, at least just by consuming bandwidth to perform scanning and transfer operations to discover new targets and infect them. In the history of malware, there have been attempts to create useful worms, that could for example fix the same vulnerabilities used to exploit a machine, by downloading and installing the appropriate patches for them; indeed the aim of such worms does not make them automatically define as “good”, because even if the goal is to favour the user of a machine, by preventing other, “evil” worms to exploit the same vulnerabilities and make more harm to the target, they still perform actions without the user’s consent, and moreover they often require to use bandwidth to download the patches, and to reboot the machine several times before the operation is complete. This is again an example of how it is hard to determine the intents of a program, even when it is possible to consider all of the aspects implied in its operation.

**Trojan horse** is a non-self-replicating malware that pretends to be performing a useful function for the user, while instead allowing unauthorized access to the machine to an external attacker. Usually the attacker would connect to the infected machine, and then perform various malicious tasks, like browsing the victim’s hard disk, recording the keys pressed, making screenshots of the desktop, up to more drastic measures like deleting files and formatting hard drives. A trojan horse need to interact with a human person to fulfill its purpose, but the attacker that connects to the target does not need to be the same person who distributed the trojan horse. Rather it is common for attackers to infect a big amount of machines so to create a “botnet” of computers ready to operate further attacks synchronously and massively at the same given time.

**Rogue security software** is a form of computer malware that deceives or misleads users into paying for the fake or simulated removal of malware, or that installs other malware. Usually it induces users to install it by showing fictitious dialogs stating that the computer is infected, and encouraging them to install (by purchasing them) some (fake) removal tool, which finally is the malicious executable.
2.2 Malware: a brief history

First viruses started to be created in the early 1970s, when ARPANET, the forerunner of the Internet, was the main and wider interconnection network available. They had the form of experimental self-replicating programs, initially ideated as jokes between colleagues in laboratories. The first virus to appear “in the wild” – that is, outside the single computer or lab where it was created – attached itself to the Apple DOS 3.3 operating system, spreading via floppy disk; it was written in 1981, and injected in a game on a floppy disk as a practical joke. Before computer networks became widespread, most viruses spread on removable media, particularly floppy disks.

In the early days of the personal computer, many users regularly exchanged information and programs on floppies. Some viruses spread by infecting programs stored on these disks, while others installed themselves into the disk boot sector, ensuring that they would be run when the user booted the computer from the disk, usually inadvertently. PCs of the era would attempt to boot first from a floppy if one was available in the drive. Until floppy disks fell out of use, this was the most successful infection strategy and boot sector viruses were the most common in the wild for many years. Nowadays similar results are seen with the spread of viruses on USB sticks, usually with the injection of an autorun.inf file, that allows the virus to automatically execute if the USB stick is inserted in an unprotected computer.

Traditional computer viruses emerged in the 1980s, driven by the spread of personal computers and the resultant increase in BBS, modem use, and software sharing. Bulletin board-driven software sharing contributed directly to the spread of Trojan horse programs, and viruses were written to infect popular software. Shareware software was equally common vector for viruses on BBS’s.

2.3 Infection strategies

In order to replicate itself, a virus must be permitted to execute code and write to memory. For this reason, many viruses attach themselves to executable files that may be part of legitimate programs. If a user attempts to launch an infected program, the virus’ code may be executed simultaneously. Viruses can be divided into two types based on their behaviour when they are executed. Nonresident viruses immediately search for other hosts that can be infected, infect those targets, and finally transfer control to the application program they infected. Resident viruses do not search for hosts when they are started. Instead, a resident virus loads itself into memory on execution and transfers control to the host program. The virus can stay active in the background and infects new hosts when those files are accessed by other programs or the operating system itself.

2.3.1 Nonresident viruses

Nonresident viruses can be thought as consisting of a finder module and a replication module. The finder module is responsible for finding new files to infect. For each new
executable file the finder module encounters, it invokes the replication module to infect that file.

2.3.2 Resident viruses

Resident viruses contain a replication module that is similar to the one employed by nonresident viruses. This module, however, is not called by a finder module. The virus loads the replication module into memory when it is executed instead and ensures that this module is executed each time the operating system is called to perform a certain operation. The replication module can be called, for example, each time the operating system executes a file. In this case the virus infects every suitable program that is executed on the computer.

2.4 Polymorphic vs. metamorphic code

With passing of time, both virus and anti-virus strategies evolved, in a more or less parallel way. One of the most important challenges anti-virus vendors and producers had to face was the creation and diffusion of self-modifying malware. The first evolution in this sense happened in 1990 with the development of the first family of polymorphic viruses: the Chameleon family. Most anti-virus software and intrusion detection systems attempt to locate code by searching through computer files and data packets sent over a computer networks. If the security software finds patterns that correspond to known computer viruses or worms, it takes appropriate steps to neutralize the threat. Polymorphic algorithms make it difficult for such software to recognize the malicious code as it constantly mutates.

2.4.1 Polymorphic code

Polymorphic code is code that uses a polymorphic engine to mutate while keeping the original algorithm intact. That is, the code changes itself each time it runs, but the function of the code (its semantics) will not change at all. Encryption is the most common method to hide the code. With encryption, the main body of the code is encrypted and will appear meaningless. For the code to function as intended, a decryption function is added to the code. When the code is executed this function reads the payload and decrypts it in main memory before executing it in turn. To gain fully polymorphic behaviour, the encryptor/decryptor pair are mutated with each copy of the code. This allows different versions of some piece of code with the same functionalities.

The way modern anti-malware products can defeat this protection is by executing the first part of a binary in a strict sandbox, allowing it to run up to the point in which the original code is fully decrypted into main memory, and then perform a static analysis on the memory itself. This way, they are ensured to deal always with the original code of the malware, which is constant amongst all variations.
2.4.2 Metamorphic code

In metamorphic binaries, with every mutation the code is rewritten in such a way to preserve its final effects in terms of interaction with the system and “middle-level” operations (i.e. the way the operations affect the system), while changing the “low-level” code (i.e. the way the operations are presented to the machine). The main difference between polymorphic and metamorphic code is that the latter does not require an explicit decrypting routine, because the code is simply restructured in a different way every time a mutation occurs. Ways in which this can be done h – but are not limited to the following techniques.

**Code scrambling** The structure of the code flow is modified so that it is not linear; various parts of the code are randomly placed in memory, and pointers referring to data and code in those sections are adjusted to ensure their validity.

**Arithmetic operations obfuscation** There exists infinite ways in which we can rewrite an arithmetical operation, without changing its meaning (and therefore, its result); let’s take as an example the addition of two numbers, x and y. If we add x and a constant k, and then y, and finally we subtract the constant from the result, we obtain a seemingly new operation, that in fact has the same meaning as the original one; in a similar way, many other procedures can be rewritten in many different ways.

**Example version 1**

```
add EAX, EBX
```

resulting machine code: 01D8.

**Example version 2**

```
add EAX, 0x01234567
add EAX, EBX
sub EAX, 0x01234567
```

resulting machine code: 056745230101D82D67452301.

**Memory addresses obfuscation** Instead of directly referencing data in memory, various offsets can be used that add up to the wanted memory address. By changing the value of the offsets, but keeping the same sum, it is possible to mask an access to a particular memory location.

**Example version 1**

```
lea EAX, [0x00112233]
call EAX
```
resulting machine code: \texttt{8D0533221100FFD0}.

\textbf{Example version 2}

\begin{verbatim}
xor EAX, EAX
add EAX, 0x00110000
add AH, 0x22
add AL, 0x33
call EAX
\end{verbatim}
resulting machine code: \texttt{31C0050000110080C4220433FFD0}.

\textbf{Register obfuscation} In modern CPUs there exist a number of general purpose registers that can be used (mostly) interchangeably. By rewriting a function and using other registers than in the original version, the code relative to it will change, without altering its meaning.

\textbf{Example version 1}

\begin{verbatim}
lea EAX, [0x22334455]
add EAX, 0x00000011
call EAX
\end{verbatim}
resulting machine code: \texttt{8D05886644220411FFD0}.

\textbf{Example version 2}

\begin{verbatim}
lea EBX, [0x22334455]
add EBX, 0x00000011
call EBX
\end{verbatim}
resulting machine code: \texttt{8D1D8866442280C311FFD3}.

In order not to over-complicate and to prevent indefinite grown of the code after a few variations have taken place, the metamorphic engine usually needs to undo the modifications made by the previous modification, and then perform the new operations on the original code. Often the code is obfuscated according to specific patterns that can happen with given probability or under given conditions, in order to differentiate as much as possible the resulting code each time. Since the metamorphic engine has to perform a variety of non-trivial tasks, it will usually have a considerable size within the module code; indeed, it often makes up to 90\% of the code of the entire binary it protects.

In case of metamorphic code, typical static anti-malware analysis techniques are useless, and also run-time memory analysis is not helping. The problem here is that the form of the code can change entirely, while its semantics is always the same. Since it is hard for detection mechanisms to perform a semantic analysis, the nearest approximation
is to identify the behaviour of the code, i.e. the interactions and the effects it has on its host system.

By observing the behaviour of a binary, we can infer with a high probability on its similarities with other classes which are well-known to be malicious, and then decide whether or not the binary is to be considered as belonging to one of them.

2.5 Sandbox environment

A sandbox is a security mechanism for separating running programs. It is often used to execute untrusted code, or untrusted programs from unverified third parties, suppliers and untrusted users.

The approach in this thesis uses a solution from Sunbelt software. CWSandbox is the name of the commercial software created by this company, targeting executable modules analysis.

The sandbox is a flexible, scalable, automated system for monitoring and reporting on the behaviour of suspect samples. It is capable to run the target module in an isolated environment, and show how the applications are executed, what system changes are made, what network traffic is generated, and any Windows API calls made.

In Windows, nearly all accesses to the system resources are done via the Windows API. The API offers functions to access the file system and the registry, to execute other applications or to install, start or stop Windows services. It also offers WinSock functions, which are normally used to communicate via TCP/IP-networks, such as the Internet. The API is implemented by different DLLs, located in the Windows system directory. CWSandbox monitors the Windows system resources including the file system, registry and other applications with special attention to communication resources. After monitoring is complete, CWSandbox writes the results of the analysis to XML files, which can be easily understood by a human, as well as parsed by another program for further analysis and/or storage.

2.5.1 XML reports structure

As we already pointed out, the data obtained from the sandbox environment is in form of an XML structure containing all of the data in an extensive and detailed fashion.

Here is an example of an entry of such XML report, relative to a file access operation:

```xml
<open_file
  filetype="namedpipe"
  srcfile="\\.\PIPE\lsarpe"
  creationdistribution="OPEN_EXISTING"
  desiredaccess="FILE_ANY_ACCESS"
  shareaccess="FILE_SHARE_READ|FILE_SHARE_WRITE"
  flags="SECURITY_ANONYMOUS"
  quantity="2"/>
```
Analyzing this entry, we can see it refers to a file access to a named pipe, whose name is “lsarpc”; we notice that the requested operation is to open an existing named pipe, with full access permissions; moreover read and write operations can also be carried out by other processes at the same time (no exclusive access); in the last position we see the number of times the operation has been executed.

A typical XML report will contain a lot of such entries, and thanks to the XML features, each entry will contain specific attributes relative to the category it belongs to – so for instance the set of attributes relative to a file access will of course be different from the ones relative to a dll load operation.

2.6 WEKA machine learning software

WEKA (Waikato Environment for Knowledge Analysis)[2] is a suite of machine learning software written in Java and freely available under the GNU General Public License, developed at the University of Waikato. It contains a collection of tools and algorithms oriented to data analysis and predictive modelling. It also offers a useful Graphic User Interface, with visualizators and graphical analysis tools. It allows the execution of several data mining tasks, such as data preprocessing, clustering, classification, regression, visualization and feature selection.

Since the software is written in Java, it can be easily ported to most of modern computing platforms, it is freely available, and contains a comprehensive collection of algorithms, preprocessing techniques and modeling techniques.

We now describe the data format associated with WEKA, which corresponds to the way we need to present our data in order to be able to use the software’s capabilities for our analysis purposes.

2.6.1 ARFF data files

ARFF stands for Attribute-Relation File Format. It is the standard input format accepted by WEKA, containing the description of the list of instances sharing a set of attributes. Each ARFF file has two distinct sections:

**header information** Contains the name of the relation, and a list with the names and the types of the attributes;

**data information** Contains the actual data, structured according to the type of attributes defined in the header section.

ARFF files allow for four types of attributes to be declared: numeric, nominal, string and date.

In ARFF files, each instance is represented on a single line, and attributes are delimited by commas – much like csv (comma-separated values) files. Usually the last field of each instance is considered by WEKA as the one that has to be predicted for new instances.
CHAPTER 2. BACKGROUND

2.7 Machine learning algorithm

In this section we describe the algorithm that has been selected in our approach in order to classify new, unknown examples within the pre-existing classes created in an initial training phase, in which a number of samples is supplied to the algorithm together with the class each of them belongs to. The purpose of the classifier is then to build a model that will correctly classify new instances in the correct classes, basing on the distance of their feature vector to the one stored in the model. The algorithm is part of the collection with which WEKA comes by default.

2.7.1 Introduction to machine learning

Storage and computation capabilities are usually considered the most important tasks performed by modern computers. Often there is the need to use recorded data relative to past (or known) events to infer something about future (or unknown) events. If this task could successfully be executed by hand by human in the past, when few examples were available, and it was quite easy to discover relations by looking at the features of the analyzed data (thanks to the small number of different parameters involved in the decision), nowadays we often have to deal with data which is considerably more numerous. In addition, it often involves handling data in very high dimensional spaces, making it difficult – if not impossible – to determine meaningful relations by humans.

Machine learning refers to the solution of this problem in the most efficient way using the memory and the computational power of modern computers. Usually in a typical machine learning related task we have an extensive set of samples, each which is known to belong to some category, and we want to determine in the most reliable way the category to which a new, untested sample belongs, by taking advantage of the information provided by the training data set. In this case the user would present the system with the original data set together with a feature which is the one to be predicted; then, when presenting a new, untested sample to the system, it should be able to successfully determine the value of the missing feature.

From now on, we will refer to the information relative to each of the samples in the training data set as exemplar, and hence the set of all the pieces of information stored by the classifier related to the training data set will be referred to as the exemplar database. We instead refer to a new, untested sample as example.

2.7.2 Instance-based learning

Instance-based learners are often referred to as “lazy”, meaning that usually little work is done when training, and most of the work is done when analyzing samples. For example, in the nearest neighbour algorithm, no work at all is done during the training phase (the items are just stored in memory, without further operations); then when a new example to analyze is provided, the whole list of items in memory is scanned in order to find the exemplar closest to it, and its category will finally be the one assigned to the example. This approach has of course major drawbacks especially in cases in
which the exemplar data set grows rapidly, and it can happen that many items don’t add any new information for the overall functioning of the learner. For example when a new item is already very close to another item of the same category, there would be no need to actually store it additionally.

A slightly different approach is the one called rule induction, meaning that as new exemplars are provided to the learner, only the ones that are incorrectly categorized by the classifier are added to the examplar database. This way, the growth of the database is fast at the beginning, when there are few samples for each class and it is likely that the newly added exemplars will be useful for refining the categorization. Thus, the size would stabilize with the passing of time. Even though this approach is in some way better than the original one, especially in case of very big data sets, there is the risk that exemplars that carry features useful for the characterisation of their class are discarded at the beginning because they are already quite similar to some other already present in the database, which in turn can have been selected just because it appeared first in the data set.

Another solution to the extremely high growth rate of the exemplar database is the use of generalised instances: when a new exemplar is presented to the learner, it is either used to refine the description for the class it belongs to, or discarded, if it would not add any information to the class. With this approach the size of the exemplars database would stabilize toward a constant value very quickly, because the introduction of new exemplars would refine the existing class description rather than being additionally stored in memory. It is anyway needed to stop the generalisation to become excessive, which can lead to loss of details relative to some particular aspects of the class. If the generalisation is done to the appropriate extent, the reduction of the size of the database should not come at the expense of the precision of the classifier.

2.7.3 Nearest neighbour

A nearest neighbour learner uses a metric to determine the distance of a new example to each of the exemplars already present in the examplar database. It will then assign the example to the class of the nearest exemplar found in memory. The training phase consists only in storing all the exemplars in memory, together with their relative classes; then, in the evaluation phase, the distance of the example is calculated relative to each of the exemplars, and the minimum is retained. It is easy to notice that when the size of the examplar database grows, the time needed to find the nearest to the example increases linearly with it, because it is needed to compute the distance function for every exemplar in the database.

2.7.4 Nearest neighbour with generalization[3]

Generalised exemplars have the property of representing more than one of the original exemplars in the training set. For our purpose, we focus on exemplars for which each feature value is actually a range of values for continuous-valued domains, or a list of values for discrete-valued domains; this way we obtain exemplars that cover a finite area
in the feature space, and specifically they have the shape of axis-parallel $n$-dimensional rectangles (hyper-rectangles). They can also be thought of as conjunctive if-then rules which are tested against the corresponding feature of the example under analysis. For each feature of a new example, if the rule is satisfied (i.e. the feature considered is within the interval in which the hyper-rectangle extends itself in the considered dimension), its distance to the class relative to that dimension will be 0. Otherwise, the distance is computed as a function of the absolute value of difference of the value of the feature of the sample and the edge of the hyper-rectangle nearest to it. The entire exemplar set can at this point be seen as a disjunctive set of conjunctive rules, with as many conjuncts as the number of features of the exemplars.

The reason of having generalization has a dual motivation:

**conceptual** it is desirable at the end of the training phase to have a model that reflects the characteristics of each of the classes in a way that is understandable by humans and properly reflects the description of the behaviour of the specific class, either with a single hyper-rectangle, or a set of them, representing the span over which each class extends into the feature space.

**practical** for performance reasons, it is not feasible to store in memory each of the samples collected for the training phase. This would cause a degradation of the performance (in term of time, or computational power) linear to the size of the database. It is more useful instead to retain in memory just a representation of each class that has been analyzed, and refine and correct it as new exemplars are added to the training set.

2.7.5 Distance estimation

A common distance function used in nearest neighbours learners is the Euclidean distance, but other types of distances can be selected for specific cases. In particular, one limit of the Euclidean distance function is that it can’t deal with missing or non-numerical attributes. To partially solve this problem, it is usually defined that the distance between a non-existing value and any other existing value is always zero, so that a missing value will not bias the overall distance that may be evaluated with other attributes. Moreover, symbolic attributes cannot be dealt with directly, so they need to be translated into appropriate numerical quantities. For instance, Boolean values representing true and false conditions could be translated into the values 0 and 1 respectively, and then the distance could be calculated in the regular way.

The algorithm outcome will be a model which holds a description of each class in terms of hyper-rectangles within the feature space, where each of the hyper-rectangles can be seen as an if-then rule.

Let $E_i$ be the i-th feature value of the example, and $H_{i}^{lo}$ and $H_{i}^{hi}$ be the lower and upper bound, respectively, of the exemplar feature value, thus representing that the hyper-rectangle relative to the exemplar extends itself from $H_{i}^{lo}$ to $H_{i}^{hi}$ relatively to the $i$-th feature.
The distance $\delta_i$ of the example to the exemplar along the $i$-th feature is then defined as follows:

$$
\delta_i^{EH} = \begin{cases} 
E_i - H_i^{hi} & \text{if } E_i > H_i^{hi} \\
H_i^{lo} - E_i & \text{if } E_i < H_i^{lo} \\
0 & \text{otherwise}
\end{cases}
$$

The final calculated distance takes into account two different weights as adjustment: the first one, $W_H$ is used to weight exemplars according to their capacity to provide a good accuracy in their prediction; the other one, $W_i$, weights each of the features according to their importance within the distance calculation, so that more important features are considered more than less useful ones. Both these weights are evaluated during the training phase, according to the way exemplars are classified during it. When analysing a new exemplar in the training set, it is first of all tried to be classified in one of the existing classes, thus its label is temporarily ignored at the beginning. If the classification made on the base of the existing rules already created corresponds to the label that the exemplar is known to have, it is considered to be correctly classified, and the weight $W_H$ relative to the exemplar that has been used for this classification is increased. This because it is likely that it will more often classify instances correctly, probably because a large number of examples lie in the vicinity of this region of the feature space. The way the weight is adjusted is that it is set equal to the ratio of the number of incorrect predictions to the sum of the number of correct precision plus the number of incorrect predictions, according to the following formula:

$$
W_H = \frac{n}{p + n}
$$

where $n$ is the number of incorrect predictions and $p$ is the number of correct prediction relative to the exemplar. At the beginning of the training phase, $p$ and $n$ are chosen so that the overall weight $W_H$ for a new exemplar is made equal to the weight of its nearest neighbour of the same class, trying to keep the area close to an exemplar as uniform as possible in case of introduction of a new one. As it appears from the formula, when an exemplar performs better, its weight will decrease, and since it is finally use as a multiplicative scaling factor for the calculation of the distance, it will finally have the effect of shortening the distance to a good exemplar, so that it will be picked up with more probability if an example is in its neighbourhood.

The rest of the formula is then a simple euclidean distance calculation.

The final forumula for calculating the distance $\Delta^{EH}$ between the example $E$ and the exemplar $H$ is thus:

$$
\Delta^{EH} = W_H \sqrt{\sum_{i=1}^{n} \left( W_i \frac{\delta_i^{EH}}{\max_i - \min_i} \right)^2}
$$

where $\min_i$ and $\max_i$ are respectively the minimum and maximum values found in the training phase relatively to the $i$-th feature, which are used to normalize all the partial distance values so they fall in the same range, and so that the weights $W_i$ relative to the features can act uniformly on them. That is, in this way scaling factors between the features are easily unbiased, and the biasing is instead left to the only action of weights.
2.7.6 Output model

As outlined in the algorithm description, the outcome of the learning phase is a model that can be used to assign a new example to one of the existing classes on which the training has been done. The model will have the form of a series of if-then rules, plus a weight for each of the classes, and a weight for each of the features. By combining all of them, it is possible to compute the distance of an example to each of the exemplars in the data base, in the same way it was explained in the previous Subsection. Then finding the nearest neighbour is a matter of finding the minimum of the distances computed against every exemplar in the data base.
Chapter 3

Automatic analysis approach

3.1 Overview

The diagram below presents the overview of the operations needed to complete the training and testing process. The process starts with the data set used for training purposes. It contains malware samples, each labeled with the class it is known to belong to. Each sample is analysed separately within the sandbox environment, and an XML report is created for each one, containing the list of operations executed by the module. The XML reports are then parsed by a Python script, whose function is to translate the list of operations contained in the reports into numerical feature vectors, creating one vector out of each XML report. At this point every sample is represented internally in the Python runtime environment memory as an array of numerical quantities, with one quantity for each selected feature. The next step is translating these vectors into a format which can be understood by WEKA, in order to proceed to the next step, which is the generation of the output model. In the workflow, this operation is executed at the end of the Python script, whose output is an ARFF file containing all the processed data. The generation of the classifier model is done via the training phase of the algorithm chosen within WEKA (which in our case is the nearest neighbour with generalization), fetched with the data output by the Python script in the previous block. We will analyze in detail each of the building blocks in the next sections.

3.2 Feature selection

From the XML report generated during the sandbox analysis process, a number of interesting features have been selected as significant for the machine learning purpose. Operations like file access patterns or network traffic are significative indicators of the behaviour of an executable, and even though they may not be very useful if considered separately, when we keep track and analyze many of them at the same time, we obtain a wider understanding of the differences in behaviour between them. The selection has been made on the basis of how significant each of the features is with respect to malicious actions that can be performed on the system, and the goal is to build a feature set that
allows to determine if the behaviour showed by different module can be considered to be produced by the same set of “semantic” instructions to some extent. With this we mean that, for instance, even though the code of two modules differs entirely between them, they may still be two different expressions of the same set of operations. This can happen for a variety of reasons, like random mutations caused by self-modifying code, or modifications made intentionally by malware writers who add features or correct bugs in the existing viruses. Each feature numerically represents some aspect (or aspects) of the behaviour of the malware, in terms of number of operations that belong to a given class of operations. For each feature, a short description of how it can exploited by a potential malware is given, in order to make the analysis more human readable, and the feature vectors somewhat meaningful of the true behaviour of the modules.

**DLL section** DLL (Dynamic-link library) is Microsoft’s implementation of the shared library concept in the Microsoft Windows operating system. This mechanism allows the operating system to keep in memory only one copy of each DLL, although the functions and data present in it can be executed and/or referenced by any process running on the system, avoiding in this way the need of loading an instance of the module for each running process that requires it. When a process wants to make use of a specific DLL, it can either specify in the header of the executable file that it is going to need it, so that the operating system can load it automatically when the process is executed, or load it at run time via specific API functions. Usually programs load DLL based on the type of functionalities they will need during the execution.

In case of slight modifications to the code of a program, as it happens in case of malware mutations, or for example when code is obfuscated via encryption or metamorphic engines, it is likely that the DLLs referenced during run time will be the same regardless of the form of the code.

Two main operations are identified in this section:

**load_image** maps a generic file into the address space of the calling process. The file can contain either code or data, and from that moment on, both can be
CHAPTER 3. AUTOMATIC ANALYSIS APPROACH

accessed as part of the program itself;

load_dll loads the specified module into the address space of the calling process. The specified module may cause other modules to be loaded in turn, if it requires so.

File system section This category is related to the most common and frequent interaction with the file system that most of the modules show. It includes file operations such as creating, reading, deleting, modifying and attributes handling. For categorization purposes, the whole set of operations that can be performed on a file has been divided in a 2 dimensional way, according to 2 different parameters: the type of operation performed, and the type of file on which it operates. In particular, for the latter we distinguish between system files (the ones contained in the “C:/Windows” folder), and user files (all the rest). Table 3.1 represents this partitioning.

<table>
<thead>
<tr>
<th>operation</th>
<th>system-space</th>
<th>user-space</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>sys_create_file</td>
<td>usr_create_file</td>
</tr>
<tr>
<td>open</td>
<td>sys_open_file</td>
<td>usr_open_file</td>
</tr>
<tr>
<td>create open</td>
<td>sys_create_open_file</td>
<td>usr_create_open_file</td>
</tr>
<tr>
<td>move</td>
<td>sys_move_file</td>
<td>usr_move_file</td>
</tr>
<tr>
<td>find</td>
<td>sys_find_file</td>
<td>usr_find_file</td>
</tr>
<tr>
<td>delete</td>
<td>sys_delete_file</td>
<td>usr_delete_file</td>
</tr>
</tbody>
</table>

Table 3.1: File system operations categories

Mutex section Mutex objects are used in the Microsoft Windows operating system to protect a shared resource from simultaneous access by multiple threads or processes. Each thread must wait for ownership of the mutex before it can execute the code that accesses the shared resource.

Related to malware behaviour, mutexes are often used to allow a single instance of malicious code at a time, usually because it usually performs operations on the system that can lead to unpredictable behaviour in case of multiple executions.

The way a malware operates in this case is to create a mutex object upon execution; if the mutex object is created and then acquired successfully, then the module is ensured to be the first and only instance to be running on the system. In case the lock on the mutex can not be acquired, the module will most likely abort every operation and terminate its thread, because another instance is already running. Other than this, viruses that attach themselves to existing executable files, often don’t want to saturate the memory with as many instances of themselves as the number of infected modules running on the system, but will likely limit the number of running malicious images to a single one.

The function of interest within this section is:
create_mutex creates or opens a named or unnamed mutex object. If the function succeeds, a handle to the requested mutex is returned.

Registry section The Windows registry is a hierarchical database that stores configuration settings and options. It contains settings for low-level operating system components as well as the applications running on the platform: the kernel, device drivers, services, SAM (Security Accounts Manager, the authentication scheme used by the Microsoft Windows operating system), user interface and third party applications all make use of the registry. The registry also provides a means to access counters for profiling system performance. The categories related to this section are listed in Table 3.2:

<table>
<thead>
<tr>
<th>operation</th>
<th>system-space</th>
<th>user-space</th>
</tr>
</thead>
<tbody>
<tr>
<td>query</td>
<td>sys_reg_query</td>
<td>usr_reg_query</td>
</tr>
<tr>
<td>enum</td>
<td>sys_reg_enum</td>
<td>usr_reg_enum</td>
</tr>
<tr>
<td>open</td>
<td>sys_reg_open</td>
<td>usr_reg_open</td>
</tr>
<tr>
<td>set</td>
<td>sys_reg_set</td>
<td>usr_reg_set</td>
</tr>
<tr>
<td>delete</td>
<td>sys_reg_delete</td>
<td>usr_reg_delete</td>
</tr>
</tbody>
</table>

Table 3.2: Registry operations categories

Process section Functions in this category are used to get information and manipulate process-related structures in the operating system. We focus our attention on 3 of them:

enum_modules is interesting to us because it allows a process to determine the presence of a debugger on the system; in some cases malware can behave differently dependent on the knowledge whether a debugger is running on the system. For example they can inhibit all the malicious operations, so that when one tries to analyze them with the use of a debugger, it will be harder to identify the exact intention of the code.

enum_processes can be used to retrieve information about the running processes on the system, and particularly the name of the executable file for the processes. It can be used by malware to determine whether an antivirus software or any other diagnostic tools are running that can be used to remove the infection.

kill_process allows a process to terminate another process. Used by malware in cooperation with the previous function to defeat anti-malware measures that could be set up on the system.

Service section In Windows, the preferred way to make a module run every time the computer is booted, is to register its executable as a service; this way it can run in the background as long as Windows is running. To do this, a module can invoke
a Windows API to register itself as a service and be ensured to be treated as such from the next time Windows is booted. Various functions allow doing this:

- **open_scmanager** used to obtain a handler to the service manager database of the local computer. It is the first step to execute if the process wants to edit the service configuration of the system.

- **create_service** creates a service object and adds it to the specified service control manager database. Contextually, automatic startup of it can be selected, so that the operating system will run the corresponding executables each time the system is started.

- **change_service_config** modifies the settings of a specified service, manipulating for example its startup behaviour, or the displayed name.

**System section** The most important function belonging to this section is the sleep function:

- **sleep** suspends the execution of the current thread until a time-out (provided as parameter to the function in terms of milliseconds) elapses. In relation with malware operation, this function is often used to perform various malicious operations at given time intervals, like capturing a screenshot of the user display every number of seconds, or regularly connect to a master server or a mail server to send periodic data resumes about the activity on the machine, together with collected data and information.

**System info section** Those functions are used to obtain different information about the environment in which the module is running:

- **get_windows_directory** returns the path to the directory in which Windows is installed (usually “C:/Windows”). Usually malware will use this information to create copies of itself and put them into the Windows directory, so that an unexperienced user will not notice its presence. Also used when the program wants to replace various operating system fundamental blocks, like the task manager or other advanced features.

- **get_system_directory** returns the path to the directory where new device and system drivers are installed by default (usually “C:/Windows/System32”). The same considerations made for the windows directory are also valid in this case.

- **check_for_debugger** checks for the presence on the machine of a system-wide debugger. Usually this is an indication that someone is analyzing or monitoring the system, and so a malware may want to disguise itself in those cases.

**Thread section** Often malware uses the *divide et impera* approach to distribute various tasks among a number of different, specified threads. This can be the case for instance of a malware module that monitors the user activity via a windows hook,
takes screenshots of the user’s desktop, opens a backdoor to allow for an attacker’s access, and periodically connects back to a server to send the data it collected. Assigning these tasks to different threads allows a better overall organization of the load, as well allowing the attacker (or the malware itself, autonomously) to dynamically activate and deactivate specific features, based on the condition of the machine or the network on which it is run.

**create_thread** creates a thread to execute within the virtual address space of the calling process. Usually a start routine is passed as a parameter to this function, so that the newly created thread will start executing the code from there.

**Virtual memory section** The virtual memory functions enable a process to manipulate pages in its and other processes’ virtual address space. In particular, here are the most important function for performing those actions:

- **vm_allocate** reserves a region of memory within the virtual address space of a specified process;
- **vm_protect** changes the protection on a region of committed pages in the virtual address space of a specified process;
- **vm_read** allows a process to read a portion of memory from the virtual address space of another process;
- **vm_write** allows a process to write onto a portion of memory from the virtual address space of another process, for which it has enough permissions; this function is widely used by malware programs to inject parts of code that will then run within the address space of other processes. This happens in case of keyloggers, or if the goal of the malware is to infect the files that are referenced by other running processes. An example can be the case of an antivirus program running on the system (that obviously did not manage to identify the malware itself when it installed on the system): by hooking onto the scanner module of the antivirus, the malware can quickly infect a large number of files as they are scanned by the antivirus, simply by inserting a function within its code.

**Window section** The functions listed in this section are related to operations with window objects. Usually they refer to graphic user interface operations, like showing or moving windows, but can also be used in other ways:

- **find_window** retrieves a handle to the window whose name matches the specified parameter. Malware can use this function to determine if a window of some particular program is open (like the task manager, or a web browser page), and then behave accordingly, like killing the process that owns the window;
- **create_window** creates an overlapped, pop-up or child window. It can be used by malware (especially rogue security software) for instance to show fake
warnings, or to display annoying windows or messages when certain conditions are satisfied.

**show_window** sets the specified window’s show state. It can be used both to hide and to show windows.

**Windows hook section** It is a common practice for keylogger to install a window- or system-wide hook in order to attach to the message queue of a process (or of the system); this way, it is easy to extract information about user interaction (especially keystrokes, that are recorded and then sent to the attacker). The most important function in this category is:

**set_windows_hook** installs an application-defined hook, in order to monitor the system for certain types of events. These events are associated either with a specific thread or with all threads in the same desktop as the calling thread.

**Winsock section** Winsock (Windows Socket 2) enables programmers to create network-capable applications to transmit application data independent of the network protocol being used.

Among all the operations that can be performed on the network, we extract one particular function, while grouping all the other together:

**gethostbyname** retrieves host information corresponding to a host name from a host database. It is used to perform a DNS translation of a host name into its corresponding IP address, that is then used to establish the actual connection with the host. Malware that collects user data can then upload the information retrieved on a ftp server or write it on an IRC channel, and in both cases it will probably need to resolve the name of the host to which it wants to connect into an usable address.

**connection** contains all of the outcoming connections initiated by the module, as well as incoming connections possibly received. Usually when analyzing software in a sandbox environment is unlikely that it will get incoming connections, since the machine is often not even reachable directly from the Internet.

### 3.3 Feature vector translation

The adopted approach has been for each XML report to analyze its entries, assigning each entry to a specific category based on the kind of operations it refers to and on a selection of its parameters, and then determine the size of each category (in terms of number of entries belonging to it). This set of numerical quantities is the representation in terms of feature vectors of the behaviour of the module.

If for some categories the translation is quite straightforward, because the parameters are not present or simply not relevant, in other cases we want to retain something from
them, without generalizing too much our description, which would let us lose some important patterns in the behaviour of a module. Yet, we don’t want the categories to capture excessively particular one from the other just because of minor differences in the values of the parameters relative to their entries, because we risk being mislead by secondary attributes that are not representative of the behaviour (like random filenames, or the number of times the same operation has been executed because of external factors that could have influenced the execution, such as absent connectivity or interrelations with other processes).

In particular, in our approach we decided to discard all attributes related to most of the classes of operations chosen, except the ones relative to file system and registry access. This has been chosen essentially according to the assumption that the most common variation that we observe from the execution of a module are related to randomly chosen filenames and registry keys. Taking a step back, this is a symptom of early evasion techniques used by viruses to mask their activity, so that for example the presence of a particular file or registry key would not reveal directly the presence of the associated infection on that system.

Still, usually also when malware uses random file or registry key names, it is likely that it will create them in the approximately same location from time to time, narrowing the choice to the last part of the element path. For this reason, we decided to basically divide all the file- and registry-related operations into two big categories each. With respect to the file system, we distinguish between operations on elements in the directory in which Windows is installed (usually “C:/Windows”), and operations in all of the other directories. With respect to the registry, we divided the operations into the ones in the branch belonging to the top level key either HKEY_USERS or HKEY_CURRENT_USER, containing information relative to the user of the computer, and the ones on all the rest of the keys, more related to the system behaviour and policies. Other than these two big distinctions (user- and system-related operations) in the aforementioned categories, a further distinction is made according to the type of operation requested (read, create, delete, move, etc., with differences between files and registry entries). This way for both categories we have a 2-dimensional division of the entire set of operations. The exact subdivision of such categories has been outlined already in the feature selection section.
Chapter 4

Testing and evaluation

4.1 Data Set

The data set used for both training and testing the machine learning algorithm has been provided by the University of Mannheim\textsuperscript{1}. They collected malware binaries over a period of three years from a variety of sources, and then assigned each of them to a category based on the analysis response given by the majority of six independent antivirus products. Then the binaries have been executed and monitored using CWSandbox, generating reports in XML format. On their website, they offer the download of the entire original data set plus the corresponding XML reports obtained with CWSandbox.

The data set consists of 3131 instances of malware binaries, covering a total of 24 distinct classes, with no less than 20 and no more than 300 instances belonging to each class. The data set contained samples belonging to classes in the quantities specified in table 4.1.

Each binary executable is named after the hexadecimal form of the MD5 checksum\textsuperscript{2} of the file itself, so that no duplicates names should occur in the data set, unless two files are actually the same file, in which case one of them can easily be detected and discarded. Appendend to the MD5 hash is the class to which the binary has been classified. Here is an example of a file named according to this rule:

\[
0219f3c1193eb8dfd4d5f5498677b3e9af63f4e0 \quad \text{LOOPER}
\]

\text{MD5 hash} \quad \text{Category}

4.2 Evaluation

Since the University of Mannheim already provides the XML reports of all the binaries, the first part of the work flow in Figure 3.1 has been skipped by us, and we used the XML reports already generated from the set. We then ran our script for data preprocessing on

\textsuperscript{1}Data set available at http://pi1.informatik.uni-mannheim.de/malheur/

\textsuperscript{2}The MD5 (Message-Digest algorithm 5) is a widely used cryptographic hash function with a 128-bit hash value
the XML reports, and obtained the ARFF data file with the feature vectors extracted from the data set, and used it for training and testing purposes.

Within WEKA, the entire set of 3131 feature vectors has been split into two sets of half this size each, of which one has been used for the training phase, and the other one for testing purpose.

Since the number of elements in the original set is odd, the split produced a set containing an item more than the other one, and expressly 1566 instances were assigned to the training set, and 1565 to the testing set. The way WEKA divides the set in this way is using a random assignment for each of the samples, at the same time trying to keep the same relative amount of samples of each category in both sets, thus avoiding situations in which most (or all) the items of a category are assigned to the same set. This way both sets are representative of the composition of the original set.
4.3 Results

4.3.1 Overall statistics

In the training phase, a model was generated in approximately 4.26 seconds, during which 29 hyper-rectangles and 4 single examples were produced, representing the 24 classes in which the binaries were categorized.

As it appears from Table 4.2, the proposed solution successfully managed to correctly categorize more than 99% of the instances analyzed. Namely, only 9 instances were incorrectly categorized out of 1565 on which the test has been performed, which provide at least a good starting point for this approach, that can be further improved with the addition of new features.

The kappa statistic is a statistical measure of agreement for qualitative items. It is usually considered more significant than simple percent agreement calculation, because it considers also the chance that an assignation can be successful by chance.

<table>
<thead>
<tr>
<th>Correctly Classified Instances</th>
<th>1556</th>
<th>99.42 %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrectly Classified Instances</td>
<td>9</td>
<td>0.58 %</td>
</tr>
<tr>
<td>Total Number of Instances</td>
<td>1565</td>
<td>100.00 %</td>
</tr>
<tr>
<td>Kappa statistic</td>
<td></td>
<td>0.9939</td>
</tr>
</tbody>
</table>

Table 4.2: Statistics summary

4.3.2 Class-specific statistics

Table 4.3 presents statistical results evaluated for every class present in the data set. This is useful to quickly determine if the classifier is performing well, or at least in a balanced way, among all the classes, and can help, in case of bad performances of the classifier, to find out whether the problem lies in a particular class which can be for example under represented, meaning that not enough samples are present for it to allow new example to be correctly assigned to it, or which is too similar to another class in the set, that is that the two classes are easily confused by the classifier. The values present in the table include:

**TP Rate** true positive rate: the fraction of instances belonging to this class, correctly assigned to it.

**FP Rate** false positive rate: the fraction of instances not belonging to this class, but incorrectly assigned to it.

**Precision** defined as the number of true positives divided by the total number of elements assigned to the class (i.e. the sum of true positives and false positives). A perfect Precision score of 1.0 means that every item that was assigned to the class was belonging to it (but says nothing about whether all the items belonging to the
class were retrieved). Calling relevant the items belonging to a class, and retrieved the ones assigned to it, we can express the Precision parameter as:

\[
\text{Precision} = \frac{tp}{tp + fp} = \frac{|\{\text{relevant items}\} \cap \{\text{retrieved items}\}|}{|\{\text{retrieved items}\}|}
\]

where \( tp \) and \( fp \) are the number of true positive and false positive, respectively.

**Recall** defined as the number of true positives divided by the total number of elements actually belonging to the class (i.e. the sum of true positives and false negatives). A perfect Recall score of 1.0 means that all the items belonging to a class were assigned to the class (but says nothing about how many items not belonging to the class were also assigned to it). Calling relevant the items belonging to a class, and retrieved the ones assigned to it, we can express the Recall parameter as:

\[
\text{Recall} = \frac{tp}{tp + fn} = \frac{|\{\text{relevant items}\} \cap \{\text{retrieved items}\}|}{|\{\text{relevant items}\}|}
\]

where \( tp \) and \( fn \) are the number of true positive and false negative, respectively.

**F-Measure** is a measure that combines Precision and Recall, in terms of their harmonic mean. It reaches its best values at 1, and worst score at 0, and it is a good single-valued numerical indicator of the accuracy of the classification. It is defined as follows:

\[
F = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}
\]

**ROC Area** specifies the probability that, when we draw one positive and one negative item at random (relative to a specific class), the decision function assigns a higher value to the positive than to the negative example.

During our tests, the classifier performed overall well on all the classes, with some non-perfect scores relative to some classes, but with general good results. Analysing some of the values in the table, we notice how the f-measure relative to the class POISON shows the lowest score within the set. If we compare this with the size of the classes within the data set, we can see that the class POISON is represented by only 26 samples, which in fact makes it the class with the least number of samples in the set. In particular, for this class we can notice how the Recall factor is 1, meaning that all the items belonging to the category were indeed assigned to it, but we have a somewhat low Precision score, which tells us that probably the model built for this class was excessively wide, leading to the inclusion in this class of items that were actually belonging to other classes. Apart from this isolated case, the rest of the classes show a convincing homogeneous high score for most of the scores.

### 4.3.3 Confusion matrix

Each column of the confusion matrix in Table 4.4 represents the instances in a predicted class, while each row represents the instances in an actual class. For a perfect classifier,
## CHAPTER 4. TESTING AND EVALUATION

### Table 4.3: Detailed accuracy by class

<table>
<thead>
<tr>
<th>Class</th>
<th>TP Rate</th>
<th>FP Rate</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Measure</th>
<th>ROC Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASINO</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ALLAPLE</td>
<td>0.98</td>
<td>0</td>
<td>1</td>
<td>0.98</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>RBOT</td>
<td>0.994</td>
<td>0</td>
<td>1</td>
<td>0.994</td>
<td>0.997</td>
<td>0.997</td>
</tr>
<tr>
<td>PODNUHA</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LOOPER</td>
<td>1</td>
<td>0.001</td>
<td>0.955</td>
<td>1</td>
<td>0.977</td>
<td>1</td>
</tr>
<tr>
<td>ADULTBROWSER</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>VAPSUP</td>
<td>1</td>
<td>0.001</td>
<td>0.955</td>
<td>1</td>
<td>0.977</td>
<td>1</td>
</tr>
<tr>
<td>VAPSUP</td>
<td>0.956</td>
<td>0.001</td>
<td>0.977</td>
<td>0.956</td>
<td>0.966</td>
<td>0.977</td>
</tr>
<tr>
<td>SPYGAMES</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SALTITY</td>
<td>1</td>
<td>0.004</td>
<td>0.667</td>
<td>1</td>
<td>0.8</td>
<td>0.998</td>
</tr>
<tr>
<td>FLYSTUDIO</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>POISON</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ZHELATIN</td>
<td>1</td>
<td>0.001</td>
<td>0.955</td>
<td>1</td>
<td>0.977</td>
<td>1</td>
</tr>
<tr>
<td>MAGICCASINO</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SWIZZOR</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PORNDIALER</td>
<td>0.962</td>
<td>0</td>
<td>1</td>
<td>0.962</td>
<td>0.98</td>
<td>0.981</td>
</tr>
<tr>
<td>VIRUT</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>EJIK</td>
<td>0.964</td>
<td>0</td>
<td>1</td>
<td>0.964</td>
<td>0.982</td>
<td>0.982</td>
</tr>
<tr>
<td>DORFDO</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Weighted Avg</strong></td>
<td><strong>0.994</strong></td>
<td><strong>0.001</strong></td>
<td><strong>0.996</strong></td>
<td><strong>0.994</strong></td>
<td><strong>0.995</strong></td>
<td><strong>0.997</strong></td>
</tr>
</tbody>
</table>

The confusion matrix should be null, except the items on the main diagonal, meaning that all the items belonging to any particular class were correctly assigned to it. The confusion matrix is useful to determine whether the classifier is performing well, and, if it is not the case, it can help to quickly have an idea of how the results given are wrong. Especially, it can easily show whether two classes are completely swapped, meaning that for example the labeling is inconsistent with the data set, or it can show whether one class is completely included in another one, meaning that they are not equally represented in the classification model.
### Table 4.4: Confusion matrix

|   | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x |
|   | 74| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 139| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 49 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 153| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 110| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

 classification as

- a = CASINO
- b = ALLAPLE
- c = RBOT
- d = PODNUHA
- e = LOOPER
- f = ADULTBROWSER
- g = VAPSUP
- h = ROTATOR
- i = BANCOS
- j = WOIKOINER
- k = VIKING_DLL
- l = LDPINCH
- m = SPYGAMES
- n = SALITY
- o = FLYSTUDIO
- p = POISON
- q = ZHELATIN
- r = MAGICCASINO
- s = SWIZZOR
- t = PORNDIALER
- u = VIRUT
- v = EJIK
- w = VIKING_DZ
- x = DORFDO

---

**CHAPTER 4. TESTING AND EVALUATION**

---

33
Chapter 5
Discussion

5.1 Conclusions

The methodology outlined in the thesis project has shown good results, in terms of reliability and correctness of the classification.

Even though there are of course limits and space for further improving, both regarding the methodology itself and the tools used (sandbox and machine learning software), the results looks promising.

Anyway, malware is constantly evolving, and it is likely that in a more or less short time the solution will need to be reviewed, because of more powerful and complex malware approaches that may be developed. We show however how, by completely ignoring the information about code and static signatures, and moving towards a dynamic analysis where the behaviour of each binary is analyzed and classified, we can obtain good results, that are independent on the particular code expression of the malware.

5.2 Future work

From the use of WEKA software, an important limit has been the impossibility to retrieve the calculated distance between an example and its nearest exemplar in the exemplar data base. This is a strongly limiting factor when it comes to analysing binary files that are in advance not known to be malware, but could potentially be regular executables. This somewhat restricts the usefulness of this technique, since with the current implementation, a file would always be classified in one of the classes, which will be the nearest, but not necessarily near enough to be reasonably considered as belonging to it. In other words, also the nearest neighbor can actually be much further away than expected, and in this case it is probably better to exclude it from the class, and ask for the intervention by a human actor, who can then consider the best way to proceed with the sample. The ideal case would be having a distance threshold for each of the classes, so that if an example is to be assigned to a class because it appears to be closest to it, but the distance is not small enough to consider it similar to the rest of the class, the example is simply left unassigned, and the user notified about this.
Moreover, since the sandbox analysis takes place during a limited amount of time, and under specific environment conditions (like network connectivity, operating system version, other software installed), it is possible that the analysed binary would behave differently if executed for a longer period of time, or under a different condition. In all of these cases, further analysis is needed, most likely with the intervention of a human actor, which will disassemble and analyze the binary to find similarities and differences with known infection patterns.

At the current time, the scope of the solution developed has to be seen more like an aid to anti-malware vendors than a solution for personal use on a user PC; this is mostly due to the large amount of time and computational requirements that makes the approach not useful for real time analysis of malware. However, it can be of significant help to anti-malware product developers, that have to deal daily with a very big number of new samples acquired “in the wild” which need to be categorised and possibly to be classified as variations of already known classes of malwares. This can speed up the work of software analysts, that can in this way focus on the analysis of new and unclassified binaries.

There is space for improvement in this sense, for example by integrating a real-time module which can dynamically build a feature vector coherent with the description given above, so that at regular intervals, or when particular operations are performed by the program, a comparison can be done with the existing exemplars data base and when a behaviour similar enough to one of the classes is observed, the process can be stopped and the user warned about the threat. In this sense, the approach would be complementary to the classic operations that are already carried out by regular antivirus scanners, meaning that there is no need to determine a behavioural trace for malwares that are easily identified by static analysis; instead, when new, polymorphic viruses are detected and analyzed by antimalware products companies, their behavioural signature could be distributed to the users and added to the exemplar data set stored on the machine.
Bibliography

