FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR'S THESIS

Title:	Static detection of malicious PE files
Student:	Jakub Ács
Supervisor:	Mgr. Martin Jureček
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2018/19

Instructions

Study previous attempts to use machine learning for static malware detection. Get familiar with the PE file format and machine learning methods. Gather various PE files (both benign and malicious) and extract features that could be useful for classification. Employ a feature selection algorithm to determine which features carry valuable information for classification. Train a machine learning model for malware detection. Using various methods, evaluate the results and compare with previous work.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague December 18, 2017

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Static detection of malicious PE files

Jakub Ács

Department of Computer Systems
Supervisor: Mgr. Martin Jureček

May 11, 2018

Acknowledgements

In the first place, I would like to thank my supervisor, Mgr. Martin Jureček for his willingness to help at every moment. His knowledge and experience in malware detection became the building blocks for this bachelor thesis. Also, my special thanks go to prof. Robert Lórencz for keeping an eye on the general progress and for his invaluable experience that he was willing to share. I am sure these will be valuable throughout my whole career, not only for this thesis. For language corrections, I am thankful to my dearest friend, Barbora Filipová and my uncle in law, Leonard Bryan. Last but not least, I would like to thank my parents for making the study at Czech Technical University possible for me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 11, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Ács), Jakub). *Static detection of malicious PE files.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Doposud známé a užívané postupy při detekci škodlivého softwaru (malwaru) přestávají poskytovat dostačující úroveň ochrany, a proto je zřejmé, že v budoucnu budou muset být nahrazeny, nebo minimálně doplněny inovativními metodami. Tato práce se zaměřuje na využití metod a algoritmů strojového učení pro detekci malwaru. Použitím statických příznaků extrahovaných ze souborů ve formátu PE, jimiž jsou například importované funkce, se nám podařilo natrénovat více modelů pro detekci škodlivých souborů. Nejlepší z modelů dosáhl téměř 95% úspěšnosti. Tento model může být použit, mimo jiné, na předběžnou eliminaci, následovanou klasickými postupy detekcí. Další využití může tato práce nalézt ve výzkumu, kde poslouží jako další z možných vstupů pro probíhající výzkum v oblasti automatické detekce malwaru.

Klíčová slova Statická analýza malwaru, Detekce malwaru, Statické atributy, Strojové učení, PE formát, Python

Abstract

Since the classical used approaches for malicious software (malware) detection are failing to provide sufficient level of protection, it is becoming clear that

these will have to be substituted or at least enhanced by new, inovative methods in the future. This thesis focuses on utilizing machine learning techniques for malware detection. Using static features extracted from the PE files like imported functions, we were able to train various machine learning models for malware detection. The best performing model reached almost 95% accuracy. This model can be used for instance, for preliminary detection of malicious PE files. Another purpose of the thesis can be found in the following research, it could serve as another input for future automatic malware analysis studies.

Keywords Static malware analysis, Malware detection, Static attributes, Machine learning, PE file format, Python

Contents

List of Figures

List of Tables

List of Listings

Introduction

Ever since computers came into existence, there has been malicious software (malware). Over the years, motivation for malware creators arose from just minor jokes to serious money or identity thefts. In late 2016, the world was left shocked by the capabilities of Mirai malware [1] to create a botnet consisting of 300,000 IoT devices with destructive power for DDoS attacks. One year later, more than 230,000 machines running Microsoft Windows OS got infected by WannaCry ransomware, [2], which misuses Eternal blue vulnerability. The ambition of ransomware (type of malware) is to collect money from the victims in return for decrypting their hard drives that were encrypted during the infection. Current anti-viral commercial products are unable to detect previously unseen malware. This is because the principle of signature-based methods that being used. To solve this problem, machine learning methods and classifiers are utilized for malware detection. In 2001, it was shown by [3], that this approach could indeed lead to successful malware detection.

The need for successful detection of malware should be clear from previously mentioned real world cases, despite these being just drops in the ocean of cyber security incidents. Another reason I chose this topic for my bachelor thesis is that am greatly interested in cyber security, but at the same time a I was curious to see it combined with machine learning. In this thesis we will utilize machine learning approaches for malware detection. Therefore, I took the opportunity to combine two fields and get a deeper insight into both malware detection problematics and machine learning.

The goal of this thesis is to train machine learning model for malware detection. After accumulating and studying previous research on this topic, we will perform a workflow of classic machine learning study. This will consist of gathering data, selecting relevant attributes, training and finally evaluating the results of our classifier. Each of these phases will be described in details. This thesis only focuses on usage of static information extracted from PE files, the thesis does not encompass dynamic analysis, nor files in other formats.

Structure of the thesis corresponds to the goal. In the first 4 chapters we

provide the necessary knowledge and current state of the art, followed by the description of our tools we developed for successful completion of this thesis and description of our approach and achieved results.

Chapter 1

Malware analysis and detection

In this chapter we will introduce the terms and concepts necessary to understand the thesis and current state of malware detection field. The first are general malware analysis approaches, followed by the ways these are used for malware detection. The final section of this chapter is dedicated to obfuscation, which is heavily used by malware writers to evade detection.

1.1 Malware analysis

In our context, we will understand malware analysis as a process of analysing the executable to help us determine if it is malicious. In order to decide about maliciousness, one needs to have a decent insight into what the executable does. To gain this insight, malware analysis methods are used. These methods are divided into three categories based on the traces that are followed and considered in the process. The division often differs amongst sources. We decided to follow the division introduced in [4].

1.1.1 Static malware analysis

Static analysis uses methods that do not require the executable to be run. These can be static information extracted from the executable such as attributes of PE (Microsoft's format for DLL's and executables, described in more details later) file or the list of functions which the executable imports, not to mention many more. Gathering this information, a malware analyst can already be able to decide if the original file is malicious or not. In case the information gathered from the structure is not sufficient, the executable needs to be reverse engineered, decomposed into small parts and analysed once more. Since we don't need to run the executable for static analysis, the extraction of the information is fast in comparison to dynamic analysis. However, there is much information, and relationships that cannot be revealed by static analysis.

1.1.2 Dynamic malware analysis

Dynamic analysis comprises of running the file in a so-called sandbox. Sandbox is an isolated environment, which is pre-built so that it can record all actions of the process. The reasons for using a sandbox environment are keeping the host machine clean while having more control over analysis. Based on analyst's observations of actions taken by the executable, his task is to evaluate the maliciousness. However, there are some drawbacks to this method. In the first place, not all malware executes its malicious activities right after being started. Therefore, these methods are time consuming in general. Other than that, there are methods for the detection of virtualized environments. Once the malware recognizes that it is being run in a sandbox, it can totally change its behaviour and act benign. Therefore, it can evade the detection by dynamic analysis. Raffetseder et al. dedicated their study to detection of emulated environments in [5].

1.1.3 Hybrid malware analysis

Both static and dynamic malware analysis methods have significant advantages and disadvantages. In order to eliminate the disadvantages, hybrid analysis methods were proposed. Any combination of static and dynamic methods can be referred to as a hybrid malware analysis.

1.2 Malware detection

Once familiar with the malware analysis methods to extract the information, we can move to decide the problem if the executable is malicious or not. This problem is called malware detection and it is in general considered undecidable. In [6], Chess and White proved this in a theoretical way. For practical usage, of course, malware detection can achieve solid results and help to protect the computer systems.

Moreover, malware is not straightforwardly defined. In [7], the definition of malware was mentioned as one of the open problems in computer virology. In [8], it was formally defined. To follow, there are infinite numbers of ways to write the same program or functionality. Or, in other words, there are an infinite number of instruction sequences, that lead to the same result. This could be achieved for instance by using the obfuscation methods described in [9] or later in this chapter.

1.2.1 Signature based detection approach

The classic approach to malware detection, used by most of the commercial AV's, is signature-based detection, [10]. This approach comprises of two phases. In the first phase, the executable is manually analysed by a human malware analyst. If he decides that the executable is malicious, signature of the malware is extracted and stored in the database. Signature is simply anything that uniquely identifies the malware. From sequence of bytes to combination of imported functions. Moreover, the way signatures are created is the most valued secret of commercial AV companies. After the extraction, the signature is stored in the database. The detection phase itself then consists of inspection of the suspicious executable and searching for any known signature that would recognize it as a known malware. This technique is commonly used by commercial AV's because of its low false positive rate (described in section 4.5). However, this method is not able to detect new kinds of malware, or even obfuscated malware, because it relies on presence of known signature. It also needs the human factor to intervene and create the signature. Another disadvantage is the dependence on frequent malware signature database updates.

1.2.2 Using machine learning for malware detection

In past years, scientists started to utilize machine learning algorithms for malware detection. Schultz et al.'s work, [3], in 2001 is generally considered the first ever usage of machine learning for malware detection. In this approach, detection model is trained by machine learning algorithms on training data and then used for detection. In comparison to a signature based approach, machine learning methods tend to have a higher false positive rate. However, thanks to

predictive models, these methods proved to be able to recognize malware that wasn't previously seen by the model and there is no need to manually analyse each sample. Therefore, until suspicious executable is properly analysed and added to signature database, machine learning models can provide a decent first level of defence.

1.3 Malware obfuscation

Obfuscation, or "*attempt to hide the original intention*", is widely used as an evasion technique by malware writers. In [9], the whole malware detection problem is viewed as an "*obfuscation-deobfuscation game between malware analysts and creators*". Every time a newly obfuscated malware occurs, malware analysts need to find a reliable way to detect it. Once they succeed, the malware creators come with a newly obfuscated malware which can evade the detection. This process repeats constantly.

1.3.1 Classic malware obfuscation techniques

Various techniques are used by malware developers to make the code look different while the behaviour remains the same. We will stick to the most known techniques mentioned in [9]. Most of these techniques are pretty much self-explaining:

1. dead code insertion
2. code transposition
3. register substitution
4. instruction substitution

Dead code insertion

This technique uses insertion of instructions that "do nothing" between effective instructions. The most basic dead code insertion is, for example, using NOP instruction to break the signature. NOP stands for no operation instruction. This is easily detected, but there are infinite numbers of ways to insert instruction sequence that leaves the state of CPU unchanged. The other basic example can be seen at Listing 1.

```
add eax, 0
sub edx, 0
```

Listing 1: Example of dead code sequences

Code transposition

Changing the order of independent instructions would be the most basic example of code transposition. However, we can go further by using small distance relative jumps. Listing 2 represents the original code, Listing 3 depicts one of the possible instruction transposition. Both jumps and independent instruction reordering was used. These two codes are equivalent, but will be represented by different machine code.

```
start:
                xor eax, eax
                mov edx, 0x58
                add eax, edx

; code continuing
```

Listing 2: Original code

```
start:
                mov edx, 0x58
                jmp l2
l1:
                add eax, edx
                jmp l3
l2:
                xor eax, eax
                jmp l1
l3:
; code continuing
```

Listing 3: code after transposition

Register substitution

Another method to break the malware signature is interchanging used registers. For instance, let's say the target function uses `eax` register as a counter and `edx` to store the result. In case we interchange all occurrences of `eax` and `edx` in the function, it will remain functionally equivalent to its original form, despite the machine code being different, therefore, the potential signature match evaded.

Instruction substitution

Thanks to the wealth of x86 instruction set, which is prevalent amongst today's PCs, there are many possibilities to find equivalent instructions. For example, using the XOR operation on register itself is equal to setting it to 0. Therefore the effect of `xor eax, eax` is equal to `mov eax, 0`, but the machine codes are different.

1.3.2 Obfuscated malware

Stamp and Wong divided obfuscated malware into three categories, based on the complexness of obfuscation, in [11]:

- Encrypted
- Polymorphic
- Metamorphic

Encrypted malware

Encrypted malware consists of two parts. Decryptor and encrypted body. Decryptor is a small piece of code, intended to run first, that is responsible for decrypting the malware body. Body is usually encrypted using some basic encryption, for instance xor-ing the bytes with different key. It is important to use different key for each infection (to make sure each body is different). However, it is relatively easy to detect encrypted malware with signature-based methods, focusing on matching the decryptor, which remains the same.

Polymorphic malware

Consisting of decryptor and body too, polymorphic malware adds another layer of obfuscation to the executable. To create polymorphic malware creators use various obfuscation techniques to change the decryptor for each infection, therefore undetectable by the plain signature based approach.

Metamorphic malware

Metamorphic malware brings the obfuscation to a whole new level. Metamorphic is defined in [11] as "*Software is said to be metamorphic provided that copies of the software are all functionally equivalent, but the internal structure differs*" As mentioned previously, there are an infinite number of ways how to write a code with the same functionality. Metamorphic malware has the ability to be completely different for every single infection, making it undetectable by signature-based detection.

Chapter 2

Related work

In the years it has become clear that with the amounts of malware being created, it is impossible to wait for each suspicious file to be manually analysed by professionals - leaving the computer systems at risk of being infected by it. We certainly need some layer to protect our computers and infrastructure from the files that have newly occurred and are yet to be analysed.

Over the past decades, numerous attempts to employ machine learning and statistical methods to malware detection problem occurred. Applying various approaches and using many different features, researchers have mostly acquired solid results. As mentioned previously, these methods could be the solution for providing a certain level of security during the time that new malicious executables are being analysed.

In [3], Schultz et. al. used 3 different classification models with the following features: strings extracted from the executable, carefully chosen byte sequences occurring in the executable, functions/DLLs gathered from the PE headers. It is believed to be the first attempt to use machine learning to solve malware detection problems. The detection rate they reached was quite impressive and hinted that machine learning methods could be efficiently used for malware detection in the future.

Recently, Kozachok [12] reached new heights with carefully crafted features - binary values, that were in some way computed from PE header contents (e.g. if number of different DLLs used by executable surpassed a threshold or not). His detection rate while using decision forest and artificial neural network rose to more than 0.992 and 0.991 respectively.

Merkel et. al. proposed a nice, simple statistical approach based on hypothesis testing in [13]. Using PE header attributes, points were assigned to each executable to measure its maliciousness (the more points, the more malicious). Then, threshold was used to determine if the executable is labelled malicious or benign. The points assignment was calculated based on conditional probability.

Chistodorescu et. al. [9] provided a unique view on malware detection as

a "*obfuscation-deobfuscation game between malware developers and analysts*". Their work produced a SAFE model which was focused on recognizing known signatures even in amongst highly obfuscated executables.

Both [14] and [15] focus on recap and description of current state in using machine learning models for malware detection.

Wang et. al. [16] focused on detection of unknown malware again. They trained another model, using SVM algorithm this time, with also comparing various feature selection methods. The features they used were binarized PE header attributes.

Belaoued et. al. [17] successfully used concatenation of name and value from the Optional PE header files to create binary features. Then used chi-square hypotheses test for feature selection. To conclude, they trained rotation forest as a classifier with very high detection rate. They also put emphasis on the importance of detection speed and feature selection's contribution.

Based on previous research we believe combining PE header attributes with imported functions might be the lightweight and efficient way to train our malware detector.

Chapter 3

PE file format description

Portable executable file format defines the structure of Windows executables and dynamically linked libraries (DLLs). It defines how the loader should map the data in memory when process is being loaded. This chapter should give you a brief idea on how files stored in PE file format are structured and how we can take advantage of static examination of PE file to detect malware. More details about the file format can be found in [18]. PE format is used to store the object files as well.

3.1 Structure of PE File

See Figure 3.1 for general overview of the PE file structure. The following sections contain descriptions of particular elements of PE formatted files.

3.1.1 DOS Header and DOS Stub

To ensure backward compatibility, each PE file starts with a DOS header to allow the DOS stub placed right after it run on DOS platform. In most PE files, DOS Stub contains code to be run in case the executable is run on MS-DOS, usually just prints a line that the executable cannot be run on that particular platform.

3.1.2 Image File Header

The file header of the PE file is actually the beginning of PE document, anything before is just to make the format compatible with the DOS platform. It contains basic information about the file that can differ among the types of files that can be stored in PE format (executable, dynamically linked library or object file). Some significant fields of this header are the time stamp that tells us when the file was created, machine, which specifies the machine that the file was compiled for, characteristics field specifies the exact type of file (executable

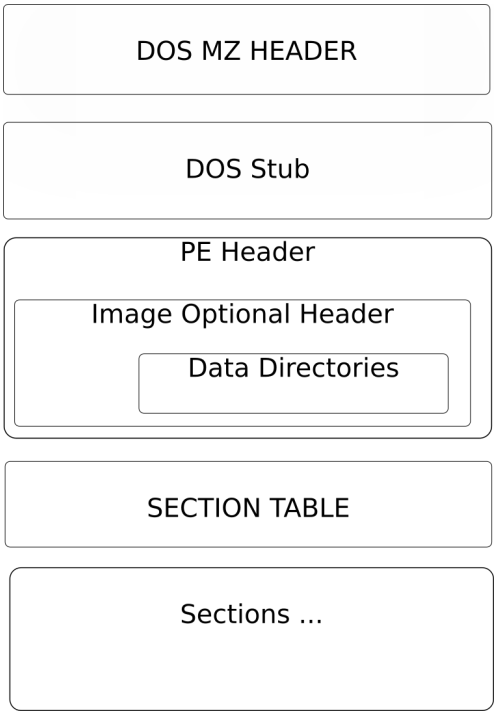


Figure 3.1: Structure of a PE File

vs dynamically linked library), number of sections that the executable contains and so on.

3.1.3 Image Optional Header

Only present in image files stored in PE file format (doesn't apply to object files). Contains various fields containing information that provides overview of the file. This information containing Subsystem version, various Flags for loader and for instance, file alignment amongst others can be valuable when distinguishing between malicious and benign files.

3.1.4 Data Directories

This special part of optional header is an array of structures that holds the addresses of particular data directories in the file. Each data directory can contain various information needed for loader. However, we are mainly interested in the `DIRECTORY_ENTRY_IMPORT` that leads us to the imported libraries and functions from these libraries. The list of functions used by a program is widely used for malware analysis. It is no surprise that the list of used functions can give us a pretty good insight on what the program actually does.

We will definitely consider using imported functions as our features when it comes to training the model.

3.1.5 Section Headers

The main part of the PE file is divided into sections to separate the code, the data and other possible sections of the program. Each of these sections has its own header that contains information about that particular section. Information like section name, size of the section, and so on are stored in these section headers. This information can hint for example on whether the file is packed, or whether it contains any other anomalies and therefore can be valuable for malware detection.

3.2 The reasoning behind choosing PE file format

We decided to choose Portable Executable format for simple reasons: Windows is the most common operating system, therefore the vast majority of malicious files worldwide are targeted towards Windows users. At the same time collecting benign executables for Windows should be fairly easy, therefore the availability of data (files in PE format) is the next reason. Another reason being the usability of methods for detection based on PE header attributes. As mentioned before, Windows is the prevalent operating system, developing working solution for protection against malicious PE files will affect most users. Third reason is the detailed documentation of this file format.

Chapter 4

Machine learning background

Since our approach for detection relies heavily on machine learning methods, we will briefly introduce and describe machine learning principles in this chapter. Machine learning in general is a very broad topic. Therefore, we will only focus on making the reader familiar with the expressions we used in this thesis. binning method

According to [19], learning in general is a process that makes the individual able to inductively generalize his observations. Of course, this generalization will always be dependent on the particular observations. Therefore, all the machine learning studies are data-dependent. When performing a machine learning study, some portion of collected data is used to "learn" or train our classifier and the other portion of data to evaluate classifier's performance. Besides data dependency of each classifier, misclassification errors can be caused by mistakes committed during learning. The two most common classification mistakes according to [20], are overfitting and bias. Overfitting of classifier leads to perfect performance on previously seen data and poor performance on unseen data. The purpose of machine learning, however is the ability to perform well on unseen data (to predict). Bias can be vaguely explained as inclination towards one particular class.

This chapter is structured as follows. In Section 4.1 differences between attribute types are explained. In Section 4.2 how to prepare these types of data for classifier training phase. Followed by Section 4.3, where selecting the relevant features is described. Section 4.4 contains description of machine learning models related to this thesis. Finally, in Section 4.5, we will describe metrics for evaluating classifiers' performances.

Please bear in mind that this chapter serves to explain the machine learning minimum needed to understand this thesis, for more information about machine learning, refer to [21], [22], [19] or many more.

4.1 Attribute data types

When collecting data, we need to distinguish between diverse types of attributes. Of course, types of attributes can be divided to various categories, in this case we divide the attribute types from a practical point of view, meaning by the different aspects of handling each attribute category.

Binomial attributes

Probably the most basic type of data is binomial attribute. This attribute can only acquire two states - true/false (alternatively 0/1 or present/absent). For instance, presence of a particular imported function in the PE file is a binomial attribute.

Nominal attributes

Nominal, or so called categorical attribute can occur in finite number of discrete states. It is important to note, that values of nominal attribute cannot be ordered in any way. For example, `OPTIONAL_HEADER.Subsystem` is a nominal attribute, despite it might appear numerical. It is necessary to realize, that each of the possible subsystem values stands for a different subsystem and besides probing for equality, we cannot do any operations between them (multiplying, ordering, ...).

Numerical attributes

Numerical attributes are attributes that are numbers. Additionally these can be divided into continuous and discrete attributes. Numerical attributes can be ordered and compared with each other, all the mathematical operations are permitted.

4.2 Data preprocessing

In the previous section we described diverse types of attributes used for model training. As the specific machine learning algorithms are working on different principles, some can handle types of data others cannot. Before training a particular model, we need to prepare the data into a form which can be handled by this model. This phase is called data pre-processing and two of the pre-processing methods will be described in this section.

Binning

Binning is a method used to discretize a continuous attribute. Let's say we have an attribute with continuous values from range (a, b). We want to divide all the possible values into n "bins". Therefore, we will define intervals for each

bin, and when the value of the attribute falls in between the margins of the bin, we take the value of the bin as the attributes value. For the equal-width binning the i-th can be defined as follows:

$$\left(a + \frac{(i-1)(b-a)}{n}, a + \frac{i(b-a)}{n}\right) \tag{4.1}$$

Equal-height binning defines the intervals to contain the same number of samples. Figures 4.1 and 4.2 depict the difference of equal-width and equal-height binning performed on the same data set.

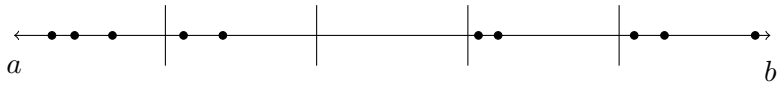


Figure 4.1: Equal-width binning

Figure 4.2: Equal-height binning

Normalization

Some classifiers may be sensitive to different ranges of attributes. Normalization is a process of scaling values for attribute to make attributes equally significant. For instance, we might want all the values of the attribute to be rescaled, so each value is a number from range $(0, 1)$. This is called rescaling and is achieved by the following transformation:

$$S'_{ij} = \frac{S_{ij} - min(A_j)}{max(A_j) - min(A_j)} \tag{4.2}$$

Where S'_{ij} is the new value of attribute A_j on sample S_i, S_{ij} is the original value, $min(A_j)$ and $max(A_j)$ are minimal and maximal value of the attribute A_j in the training data set. However, there are problems with rescaling. For instance, if enormous S_{ij} exists, this value suppresses the significance of other possible value for this attribute. Standardization is immune against this:

$$S'_{ij} = \frac{S_{ij} - \bar{A}_j}{\sigma j} \tag{4.3}$$

Where $\bar{A}_j$ is the mean and σj is the standard deviation of values of attribute A_j from training set.

4.3 Feature selection

Once the data is pre-processed, the next step is to select those features (or attributes) from the samples that are relevant for classification. In other words, selecting subset of the attributes with the best decision ability. This step is rather difficult, as stated in [23], the number of possible feature subsets of size n selected from set of size m is

$$\frac{m!}{(m-n)!n!} \tag{4.4}$$

Since the metric for evaluating the quality of given subset may not be trivial as well, this number is more than can be satisfactorily computed even for quite low n and m. Therefore, various heuristical approaches were introduced to solve this problem.

Uni-variate feature selection

Uni-variate feature selection evaluates each feature individually, without considering any relationships with other features. The most straightforward univariate feature selection method is Best individual N. The quality measure for attribute A, $Q(A)$, is computed for each attribute individually and the N best-performing attributes are selected. Plenty of different methods can be found in [23].

Multi-variate feature selection

In multi-variate feature selection, multiple features are evaluated at the same time. These methods measure the added value of particular feature subset for classification. The most straightforward multi-variate selection would be training the classifier with each feature subset. However, as depicted earlier, this approach would be too computationally difficult. In [23], several methods are introduced. One, for instance, is Sequential forward selection. Having attribute quality function $Q(A)$, where A is a subset of attributes, Sequential forward selection adds attributes to the A one at a time. Starting with empty A, in each step, the attribute $A_i \notin A$, which yields the highest value of quality function $Q(A + \{A_i\})$ is added to the A subset. The number of steps is either set to a constant, if a particular number of attributes is desired, or the procedure is repeated until the quality is rising.

Another, widely-used method is Principal Component Analysis (PCA). In PCA, new attributes are computed as the linear combinations of original attributes to provide more information in less attributes. Refer to [23] for more information.

4.4 Trained models

In this section, we will briefly describe the models that we used in our approach for detection.

4.4.1 Decision tree

Decision tree is described as one of the most simple classifiers [21]. Introduced by Quinlan, in [24], it has been used for a fairly long time. It is a hierarchical tree structure containing conditions in inner nodes and decisions in leaf nodes. When a sample is being classified by a tree, the condition for node it reaches is evaluated and based on the result, the corresponding edge is followed. Whenever a leaf node is reached, the classification process is over, and the sample is assigned a class belonging to given leaf. You can see an example of a single decision tree at Figure 4.3. We will discuss the algorithms for constructing decision trees and how to use them for classification.

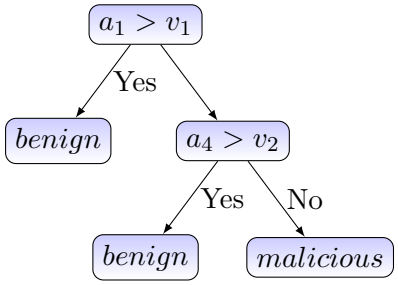


Figure 4.3: Simple example of trained decision tree used for detection

Training phase

The goal of the training phase is building the decision tree. The most classical training algorithms are ID3 [24], C4.5 [25] and CART [26]. All these algorithms work on the same principle.

Starting with available attributes set $A = \{A_1, A_2, \ldots, A_n\}$, we choose the attribute A_i that is able to provide the best split (has $max(Q(A_i))$ amongst the available attributes), remove it from available attributes $A = A \setminus \{A_i\}$ and create a node with the best split condition. One of the differences between the algorithms is their ability to perform multi-split or just binary split. Then descend to children of this node and recursively repeat this procedure. Simply choosing the attribute that can best divide the training set samples into classes each time. This "best split" is evaluated by various metrics mentioned later.

Tree training process can be stopped either when no more attributes for splitting are available, or when all the leaf nodes contain samples of one class. The latter case, on most occasions means overfitting of the tree to the training

data. Imagine the edge case where each of the leaf nodes is corresponding to exactly one sample of the training set. To avoid these problems, various methods called pruning are used. The tree can be either pre-pruned or post-pruned. In pre-pruning, we would define some simple rules to follow while building the tree, e.g. minimum number of samples corresponding to a certain node to permit further splitting. An even more straightforward example of pre-pruning could be simply setting the maximum depth of trained tree. In post-pruning, the tree is built into its full size and then simplified.

During training of a decision tree, best-split metric is needed. Most of these metrics are based on entropy, defined in 4.5.

$$H(A) = -\sum_{i=1} P(a_i) \log_2 P(a_i) \tag{4.5}$$

Where A is attribute, a_i are possible values for attribute A and $P(a_i)$ is the probability that attribute A contains value a_i.

One of the metrics used for determination of best split is information gain [27]. For binary attribute A and given set of samples S, Information gain, $IG(S, A)$ is defined as follows:

$$IG(A, S) = H(S) - \left(H(S_1) \frac{|S_1|}{|S|} + H(S_0) \frac{|S_0|}{|S|} \right) \tag{4.6}$$

Where S_i is the subset of S with value i on attribute A and $|S|$, $|S_i|$ are the sizes of these sets.

Another metric for best split is for instance gini criterion [23].

Decision phase

Example from Figure 4.3, let's say we have PE sample $S = (a_1, a_2, a_3, a_4)$, where a_1 to a_4 are extracted attributes of the PE file. v_1 and v_2 being the threshold constant values learnt in the training process. First, the condition in the root node is evaluated. If the result is true, the file is labelled as benign and the classification is over. On the other hand, if it's false, the process continues to the other node, where another condition is evaluated. The class is then chosen accordingly.

4.4.2 K-Nearest Neighbours

K-nearest neighbours (K-NN) is another of the simpler classifiers. The training phase is fairly simple and consists of storing the training dataset. K-NN classifier is defined by the training set and distance metric, that is computed to find the nearest neighbours and the parameter K, which defines the number of the neighbours considered in classification. The classification process consists of computing the distance of classified sample from all other samples stored during the training phase.

Training phase

Training phase in K-NN classifier consists of storing all the training samples in the data structure. The most straightforward approach is storing all the data in a list. A more advanced method is to use tree structures that make the decision phase faster, eliminating computing distance from samples that are visibly more distant than those already compared.

Decision phase

Decision phase consists of computing the distance between classified sample and all the stored sample to find the closest samples from the training set. Or in other words, the most similar. Since it is not always trivial to decide, how "similar" two samples are, various distance metrics are defined. One of the most widely used is Euclidian distance. When x and y are samples with n attributes and x_i and y_i are the values of i-th attribute on sample x and y respectively, Euclidian distance between x and y is defined as follows in equation 4.7.

$$D_e(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{4.7}$$

To measure distance between two samples with both nominal and numerical attribute types, for instance, heterogeneous distance metric like HVDM [28] can be used. Many more distance metrics can be found in [28].

In the standard version of K-NN classifier, once K nearest neighbours are identified, classic voting is performed, meaning the weight of each of K neighbours is equal. To prioritize the neighbours that are closer to classified sample, weighted voting is used. Vote of each neighbour can be weighted by the inverted value of the distance of a particular neighbour from the classified sample.

4.4.3 Custom classifier

We also decided to use our custom classifier inspired by [13]. We need to state that only binary attributes are considered for this classifier. A score is assigned to each classified sample, based on the conditional probabilities of presence of attributes amongst the classes in training set. The score is then compared to threshold value and if it is surpassed, the sample is labelled as malware. This provides good control over FPR and FNR results. The quality $Q(A_i)$ for each attribute A_i is computed as follows.

$$Q(A_i) = max\left(P(M|A_i), P(B|A_i)\right) \tag{4.8}$$

Where $P(M|A_i)$ and $P(B|A_i)$ is conditional probability that file from testing set with attribute A_i is malware and benign, respectively. The quality is then

corrected as follows:

$$Q(A_i) := -Q(A_i) \quad if \quad P(B|A_i) > P(M|A_i) \tag{4.9}$$

Therefore, if the attribute presence stands for benignity of the file, the quality is negative, to decrease the score of the sample and lean it more to being benign.

For imported functions only malware probability was considered, $Q(A_i) = P(M|A_i)$. In testing phase, for each tested sample S, score was computed as following:

$$Score(S) = \sum_{i=1}^{n} Q(A_i) * S(A_i) \tag{4.10}$$

Where $Q(A_i)$ is the previously defined quality of $i-th$ attribute, $S(A_i)$ is the value of $i-th$ attribute on sample S. Then the score is compared with threshold value T, and if the surpassed the threshold $Score(S) >= T$, it was labelled as a malware. Otherwise, it was considered to be benign.

4.5 Performance evaluation

Once the model is trained, it is tested on the testing data set and the results are recorded. Several metrics are used to evaluate how well the classifier performed on given testing data set. In this section we will describe those that we used in our performance evaluation.

FP, FN, TP, TN

There are four possible results of detection process:

- True positive (TP) - Malicious PE file correctly classified as malware
- True negative (TN) - Benign PE file correctly classified as benign
- False positive (FP) - Benign PE file incorrectly classified as malware
- False negative (FN) - Malicious PE file incorrectly classified as benign

		classified as	
		Malicious	Benign
in reality	Malicious	TP	FN
	Benign	FP	TP

Table 4.1: Exchange matrix

Table 4.1 sums up these results for better understanding.

From these values we can compute false positive (FPR) and false negative rate (FNR). False positive rate is ratio of benign files that were incorrectly labelled as malware to all benign files from our set, whereas false negative rate is ratio of malware files that were correctly classified to all malware files in our testing set. This is depicted at equation 4.11 and equation 4.12 respectively.

$$FNR = \frac{FN}{TP + FN} \tag{4.11}$$

$$FPR = \frac{FP}{FP + TN} \tag{4.12}$$

Accuracy

Another evaluation metric is called accuracy(ACC). Accuracy provides a general reliability of classifier. It encompasses all the files, not just those of one particular class, as was the case with previous metrics. Accuracy is ratio of correctly classified files to all files.

$$ACC = \frac{TP + TN}{TP + FN + TN + FP} \tag{4.13}$$

Chapter 5

Tools

As a programming language for our study, we chose Python, as it comes with several modules for PE files parsing as well as machine learning modules. We profited from these modules and Python's simplicity. However, we also had to write our own tools to serve our particular purpose. In this chapter we will describe the most important tools and python modules we used as well as the tools we built.

5.1 Used tools

This section contains description of the third-party tools that we benefited from.m

5.1.1 Python pefile module

Python `pefile` module, [29], is a python module used to parse files in PE format. Module loads all attributes of a PE file into an object. These attributes are then easily accessible via this object. Our custom-built tool in python, `extractor.py`, relies heavily on features of `pefile` module. Listing 4 shows how to load file in PE format to python object and access some basic attributes.

```
import pefile
pe =  pefile.PE('/path/to/file_in_pe_format.exe')

pe.OPTIONAL_HEADER.AddressOfEntryPoint
pe.OPTIONAL_HEADER.ImageBase
pe.FILE_HEADER.NumberOfSections
```

Listing 4: Parsing PE file in `pefile` module

5.1.2 Jupyter notebook

Jupyter notebook, [30], is a web-based application that enables structuring and re-running of chunks of code in a user-friendly way. We used it for the data pre-processing and classifying phases.

5.1.3 pandas library

Pandas library, [31], is a python module widely used in the data science community. It provides functionality to encapsulate data manipulation, preparing data frames for training phase that is provided by scikit-learn.

5.1.4 scikit-learn library

Scikit-learn, [32], is python framework for machine learning. Encompassing feature selection methods, classifier algorithms, validation and performance evaluation functionality.

5.2 Custom built tools

This section contains introduction and description of our custom written tools to utilize the comfortable extraction of PE file attributes.

5.2.1 extractor.py

`extractor.py` is a command line tool responsible for extracting features from PE files. It is able to extract PE header attributes, imported functions and section attributes. Whole tool consists of three self-sufficient modules, each is responsible for extracting particular attributes. The extractor.py supports two modes of extraction. Either extracting all the section names/functions and computing the number of occurrences for each — the counter mode — or extracting particular attributes that are given to the extractor.py via configuration file according to the format depicted in Listing 5 — extracting mode. Example of command with arguments to invoke the `extractor.py` can be found in Listing 6.

When invoked in counter mode, directory is expected as an input argument. Extractor runs through each file in the directory and uses one of the sub-extractors to extract all the functions (or section names, depending on the option). Maintaining the data structure holding the number of occurrences for each function (section names, respectively). The data structure is then written in csv format into specified output file. Simplified example is in Table 5.1

If the extracting mode is used, the configuration file is given as an argument. The input can be either file or directory. Extractor then uses sub-extractor modules to extract specified functions, section header attributes and

```
[HEADERS]
OPTIONAL_HEADER.Magic
OPTIONAL_HEADER.MajorLinkerVersion
OPTIONAL_HEADER.MinorLinkerVersion
OPTIONAL_HEADER.ImageBase
FILE_HEADER.TimeDateStamp
FILE_HEADER.SizeOfOptionalHeader
FILE_HEADER.NumberOfSections

[SECTIONS]
PointerToRawData
Characteristics

[FUNCTIONS]
kernel32.dll.removedirectorya
kernel32.dll.createdirectorya
user32.dll.loadbitmapa
shell32.dll.shellexecuteexa
kernel32.dll.enumcalendarinfoa

[MAX SECTIONS]
2
```

Listing 5: Format of configuration file for extractor.py

	user32.dll.loadbitmapa	shell32.dll.shellexecuteexa
count	74	53

Table 5.1: Sample extraction

PE header attributes and writes them in the output file. Simplified example of how the extracted data looks like can be seen at Table 5.2

Name of File	NumberOfSections	shell32.dll.shellexecuteexa
md5string	4	1
md5string	11	0

Table 5.2: Sample extraction

5.2.2 file_gatherer.py

This script arose for clean set gathering. It searches given source directory for PE files(recursively), computes the MD5 hash of each PE file and than copies

```
./extractor.py --config config_file.txt input_directory output.csv
```

Listing 6: Invoking extractor from the command line

the file renamed according to its MD5 hash (to avoid duplicates) to target directory.

5.2.3 writer.py

Writer.py is module used in extractor.py for encapsulating the action of writing data to the output file. Its modularity allows to easily add different output format processing.

Chapter 6

Our approach

In this chapter we will describe our approach and continuation of our steps resulting into evaluation of performance of our trained models.

6.1 Workflow

Brief workflow of our classifier training is depicted at 6.1. It consists of classic machine learning phases. Each of the phases is described in more details in individual sections.

Figure 6.1: Proposed workflow diagram

6.2 Data Gathering stage

Obtaining data is the first step, our training set consisted of 1469 benign and 1361 malicious PE files. Test set contained 367 benign and 12251 malicious executables.

6.2.1 Malware dataset

Getting malware turned out to be easier than obtaining clean set. Discovering Virusshare.com malware database, we asked for access via email, explaining that the malware will be used for research and as a part of bachelor thesis. We were granted the permission to enter the malware database. Malware is organized in archives per 64k files. However, the archives contained all sorts

of malware files. Therefore, we needed to filter the PE executable files. This was done by another simple script we wrote.

6.2.2 Clean dataset

We decided to scrap working windows image of a computer for all the executables available. The script used for this was described earlier, in Section 5.2.2. Gathering clean set with this tool consisted of the following steps:

- mounting the disk partition containing Windows 10 installation
- running the script, specifying the source and target directory for extraction

To avoid duplicates and inconsistencies, the files are saved with names generated by python hash utility using the MD5 hash algorithm. This also provides some sort of randomness to the order of the files. Despite it being totally deterministic, a human is not able to guess what a particular executable does, once it has a name being a MD5 hash, whereas the original name of the executable could provide a good hint.

6.3 Extracting features from gathered PE files

After we successfully gathered clean and malicious PE files, we needed to extract features that are significant for malware detection. To extract these attributes from a PE file, we chose to build our own python modules built on `pefile` library. These modules are described in Section 5.2.1.

6.3.1 Extracting header attributes

Extraction of PE header attributes was done in two steps. The first step comprised of extracting all the PE header attributes from all the files in our training data set by our `extractor.py` tool. Relevant PE header were then selected, as described in Section 4.3. In the second step, only relevant headers were extracted and new data sets for classifier training were created.

6.3.2 Extracting imported functions

The extraction of imported functions is done by our custom tool, extractor.py. In the first phase, we use it to extract all functions from all files in our training dataset, to create a nice overview on what functions our dataset contains. With knowing the counts of occurrences for particular functions in malware and benign executables respectively, we selected functions with the best distinguishing ability (See more in section 6.4.1). In the second phase, where the names of the functions that are to be extracted are already known, we specify

the names of the functions in a configuration file in correct format and use extractor.py to extract these according to the configuration file.

6.4 Feature selection

The following section describes how we selected our features for classification. We used different approach for selecting either functions and PE Header attributes, as each of these attributes have a different nature.

6.4.1 Selecting functions

First, we extracted the number of occurrences in malicious and benign executables for each imported function spotted in our training set using our `extractor.py` tool. Then we used jupyter notebook with python and pandas library for the next steps. Then, excluding imported functions with less than 50 total occurrences in our training set to eliminate the functions that are rarely seen. To choose the most relevant imported functions for classification, we decided to rate each function's quality. Quality being function's ability to distinguish between benign and malicious files. We computed each function's quality using conditional probability as follows: (This approach is not considering correlation between the functions and is similar to the one used in [13])

First, we computed the conditional probabilities that a file importing function f_i is benign and malicious, respectively:

$$P(M|F_i) = \frac{P(M \cap F_i)}{P(F_i)} = \frac{\#malicious\ executables\ importing\ f_i}{\#all\ executables\ importing\ f_i} \tag{6.1}$$

$$P(B|F_i) = \frac{P(B \cap F_i)}{P(F_i)} = \frac{\#benign\ executables\ importing\ f_i}{\#all\ executables\ importing\ f_i} \tag{6.2}$$

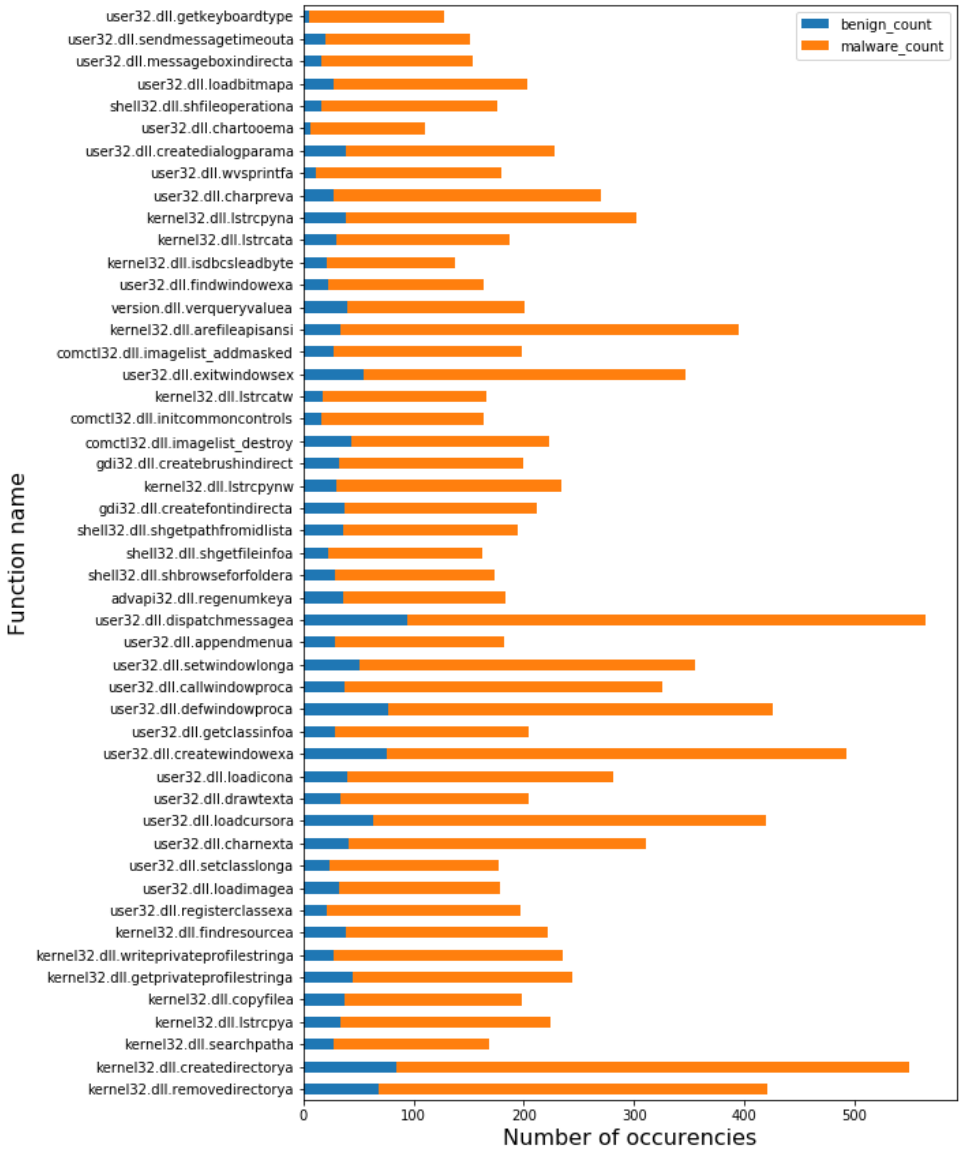
Where M is set of all malware executables, B is set of all benign executables and F_i is set of all executables containing function f_i in our training set. Then, $P(M|F_i)$ is probability that executable importing function f_i is malicious, or number of malicious files containing f_i divided by the total number of files containing f_i in our training set.

The quality for function f_i will simply be the conditional probability that executable containing particular function is a malware.

$$Q(f_i) = P(M|F_i) \tag{6.3}$$

The best performing functions (with quality higher than threshold value) are depicted at the Figure 6.2.

Figure 6.2: Function quality



6.4.2 Selecting PE Header attributes

When considering the assets provided by the PE header attributes, we looked at each attribute separately. In other words, we used the univariate feature selection approach once again. The idea behind is the same as when selecting functions. For each attribute we computed its ability to distinguish between malicious and benign executables. We called this number the attribute's quality.

First, it was necessary to prepare our data in a form suited to our needs. We computed a table for each attribute, with rows being the possible values (that occurred in our train sets) for a particular attribute and columns depicting the count of malware, resp. benign executables matching the particular value for that attribute.

This table for `OPTIONAL_HEADER.MajorSubsystemVersion` is depicted in Table 6.1

value	malware occurrences	benign occurrences
1	3	0
3	4	0
4	195	576
5	880	770
6	344	8
10	43	0

Table 6.1: Dataframe for `OPTIONAL_HEADER.MajorSubsystemVersion`

In case any attribute had more values than 20, we decided to use binning method to reduce the number of possible values to 20. Therefore, we created 20 possible values, each representing some interval into which the original value of the attribute fell. We used the equal width approach for the binning, meaning all the intervals covered same width. Binning is described in Section 4.2 in more details.

The quality for each attribute was then computed on our method based on entropy. Detailed description is following. We computed entropy for each possible value of examined attribute. Having attribute A with n possible values $\{V_1, V_2, \ldots, V_n\}$ we defined the entropy of i-th possible value:

$$E_i = -\left(\frac{M_i}{T_i}\log_2\frac{M_i}{T_i} + \frac{B_i}{T_i}\log_2\frac{B_i}{T_i}\right) \tag{6.4}$$

Where M_i is the number of malware executables with value V_i on examined attribute A. B_i is the number of benign executables with value V_i on examined attribute A. Finally, T_i is the number of all executables with value V_i on

examined attribute A. Therefore, $T_i = M_i + B_i$. We defined the quality $Q(A)$ of attribute A as follows:

$$Q(A) = \sum_{i=1}^{n}(1 - E_i)\frac{T_i}{total} \tag{6.5}$$

Where n is number of possible values for examined attribute A, E_i is the previously defined entropy and *total* is the number of all samples in our training set.

We computed the quality of all attributes and selected all that surpassed our chosen threshold. You can see how, for instance, attribute representing the number of sections in a particular file performed on figure 6.3.

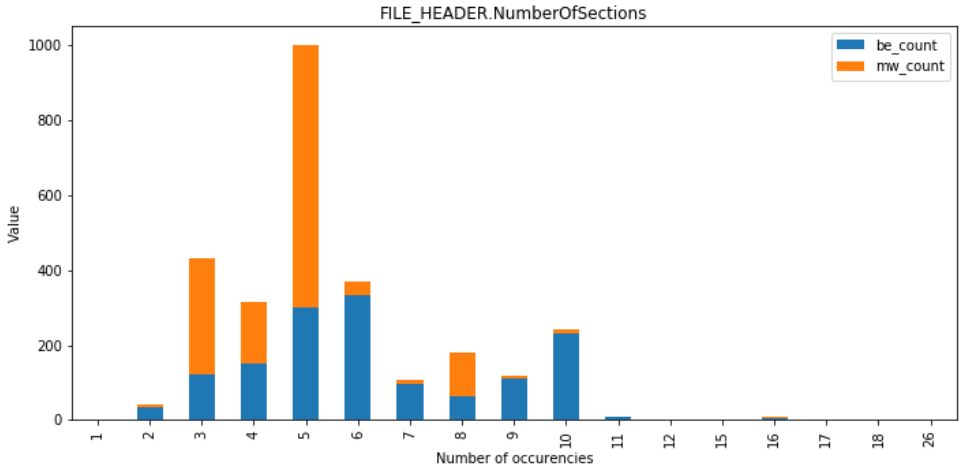


Figure 6.3: `NumberOfSections` amongst training set, computed quality: 23.62

6.4.3 Additional selection

After we computed the final list of attributes to extract from PE files, we ran our `extractor.py` to produce `.csv` for each data set(benign training, malware training, benign testing and malware testing). This `.csv` was then imported by `Jupyter Notebook`, where we performed some other corrections:

1. Binarizing categorical attributes
2. Further, more restrictive attribute selection

We added the option of more restrictive attribute selection to be able to evaluate the performance of classifiers trained on attribute sets containing both larger amount and smaller amount of attributes.

Binarizing categorical attributes

The classifiers we used are mostly treating data as numerical. Binary attributes are compatible with numerical, categorical are not. Therefore, we had to process categorical attributes as following: for each categorical attribute A with n possible values $\{v_1, v_2, \ldots, v_n\}$, n new attributes were created. Each new attribute was a binary attribute, indicating if $A = v_i$.

Further attribute selection

Finally, we chose to use two different attribute sets of different size. Both attribute sets were chosen from previously selected numerical PE header attributes, binarized PE header attributes and imported functions. The numerical attributes were selected previously, we just conducted more restrictive selection on binarized PE attributes and imported functions. Selection was based on conditional probability. For each attribute set, we chose threshold values for conditional probability and minimal number of occurrences for given attribute in training data set (all the evaluated attributes were binary, indicating the presence). Threshold for quality and threshold for occurrences differed for binarized PE header attributes and imported functions. With T_a as threshold for attributes and $O_a$for the minimal occurrences, attribute was chosen if $P(M|a_i) >= T_a$ or $P(B|a_i) >= T_a$ and more than O_a occurrences on training set were found. For imported functions, threshold values T_f and O_f were used. We then chose functions that met the following conditions: $P(M|f_i) >= T_f$ and more than O_f occurrences on training set.

6.5 Classifier evaluation

In this section, we will be evaluating various models for detection, with previously extracted attributes. Each subsection contains the results reached with each model. We considered two different set of attributes for each model. We used our testing set containing 367 benign files and 12251 malware files. Therefore, the accuracy will be skewed by the amount of malware files and will actually be rather close to true positive rate.

Decision tree classifier

The decision tree classifier acquired the best results among the models we tested. We decided to pre-prune the tree with the following parameters. The minimal number of samples for splitting to continue was set to 5. The maximal depth of the tree was restricted to 23.

K-NN classifier

For K-NN classifier, described in Section 4.4.2, we chose weighted distance metric for voting, meaning the samples from training set that were closer to classified sample had bigger impact on the classification. The number of neighbours considered was 5.

Custom classifier

Described in details in section 4.4.3, we used the classifier with three different threshold values. Each of the values was set to get results with either equal FPR and FNR, or low FPR or low FNR. In Table 6.2, results for all three thresholds are shown. We used this classifier with different threshold values, T, described in Section 4.4.3. Classifier with thresholds set to have low FNR or FPR could be used as pre-elimination for decision tree classifier.

	Original feature set		Reduced feature set	
Classifier	FPR	FNR	FPR	FNR
Decision Tree	**2.18%**	**5.48%**	2.99%	5.57%
K-NN	8.45%	9.53%	8.45%	9.61%
Custom (T=1.9)	8.72%	19.22%	5.99%	15.77%
Custom (T=-1)	34.33%	0.41%	40.60%	0.16%
Custom (T=10)	2.18%	74.60%	0.82%	83.00%

Table 6.2: Classifier performances

Comparison

With our approach, our results are similar to those achieved by Schultz, in [3] with Multi-Naive Bayes classifier. Our FPR, 2.18% is better than their proposed 6.01%, but on the other hand, we achieved higher FNR, 5.48%. The results of custom classifier based on Merkel el. al's [13], can be compared to their study. The results are not significantly better, nor worse. Both studies support the claim, that this approach can choose between having a good FNR or good FPR, but having good values for both at the same time is unlikely. However, it is not always straightforward to compare machine learning studies performed on different data sets.

Conclusion

The goal of this thesis was to study previous research on usage of machine learning methods for malware detection, to gather enough data in PE format to perform our own machine learning study, to employ feature selection methods to select the relevant attributes for classification, and finally, to train our own classifier that can be used for malware detection.

In the first step, we performed state of the art research consisting of summarizing various sources of literature containing books, scientific articles and documentation of PE format and python libraries. After that, we successfully gathered more than 15 000 PE files. Followed by writing our own tools for attribute extraction and then performing the feature selection before training our own classifiers for malware detection. Besides this, we tried various machine learning models and evaluated the results. Thus, meeting the requirements of the task of this thesis.

Our classifier confirmed, just like the previous research, that usage of machine learning approaches for malware detection can be successful. However, in the commercial sphere, the FPR needs to be lower than the one achieved by machine learning models. Therefore, combining traditional signature matching with new methods started to be encompassed by the commercial AV's. This thesis contains relevant information for future research and the classifier trained can be used for malware detection.

Bibliography

[1] Antonakakis, M.; April, T.; et al. Understanding the Mirai Botnet. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, 2017, pp. 1093–1110.

[2] Ehrenfeld, J. M. Wannacry, cybersecurity and health information technology: A time to act. *Journal of medical systems*, volume 41, no. 7, 2017: p. 104.

[3] Schultz, M. t. G.; Eskin, E.; et al. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, IEEE, 2001, pp. 38–49.

[4] Damodaran, A.; Troia, F. D.; et al. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, volume 13, no. 1, Feb 2017: pp. 1–12, ISSN 2263-8733, doi:10.1007/s11416-015-0261-z.

[5] Raffetseder, T.; Kruegel, C.; et al. Detecting System Emulators. In *Information Security*, edited by J. A. Garay; A. K. Lenstra; M. Mambo; R. Peralta, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, ISBN 978-3-540-75496-1, pp. 1–18.

[6] Chess, D. M.; White, S. R. An undetectable computer virus. In *Proceedings of Virus Bulletin Conference*, volume 5, 2000.

[7] Kramer, S.; Bradfield, J. C. A general definition of malware. *Journal in Computer Virology*, volume 6, no. 2, May 2010: pp. 105–114, ISSN 1772-9904, doi:10.1007/s11416-009-0137-1.

[8] Filiol, E.; Helenius, M.; et al. Open Problems in Computer Virology. *Journal in Computer Virology*, volume 1, no. 3, Mar 2006: pp. 55–66, ISSN 1772-9904, doi:10.1007/s11416-005-0008-3.

[9] Christodorescu, M.; Jha, S. Static Analysis of Executables to Detect Malicious Patterns. Technical report, WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES, 2006.

[10] Mujumdar, A.; Masiwal, G.; et al. Analysis of signature-based and behavior-based anti-malware approaches. *International Journal of Advanced Research in Computer Engineering and Technology*, volume 2, no. 6, 2013.

[11] Wong, W.; Stamp, M. Hunting for metamorphic engines. *Journal in Computer Virology and Hacking Techniques*, volume 2, no. 3, 2006: pp. 211–229, ISSN 1772-9904, doi:10.1007/s11416-006-0028-7.

[12] Kozachok, A.; Kozachok, V. Construction and evaluation of the new heuristic malware detection mechanism based on executable files static analysis. *Journal of Computer Virology and Hacking Techniques*, 2017: pp. 1–7.

[13] Merkel, R.; Hoppe, T.; et al. *Statistical Detection of Malicious PE-Executables for Fast Offline Analysis*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ISBN 978-3-642-13241-4, pp. 93–105, doi:10.1007/978-3-642-13241-4_10.

[14] Nath, H. V.; Mehtre, B. M. Static malware analysis using machine learning methods. In *International Conference on Security in Computer Networks and Distributed Systems*, Springer, 2014, pp. 440–450.

[15] Souri, A.; Hosseini, R. A state-of-the-art survey of malware detection approaches using data mining techniques. *Human-centric Computing and Information Sciences*, volume 8, no. 1, 2018: p. 3.

[16] Wang, T.-Y.; Wu, C.-H.; et al. Detecting unknown malicious executables using portable executable headers. In *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, IEEE, 2009, pp. 278–284.

[17] Belaoued, M.; Mazouzi, S. A real-time pe-malware detection system based on chi-square test and pe-file features. In *IFIP International Conference on Computer Science and its Applications_x000D_*, Springer, 2015, pp. 416–425.

[18] Microsoft corporation. Microsoft Portable Executable and Common object file format specification. 2010.

[19] Shalev-Shwartz, S.; Ben-David, S. *Understanding Machine Learning: From Theory to Algorithms*. 2014.

[20] Hastie, T.; Tibshirani, R.; et al. *The Elements of Statistical Learning*. Springer Series in Statistics, New York, NY, USA: Springer New York Inc., 2001.

[21] Alpaydin, E. *Introduction to Machine Learning*. The MIT Press, second edition, 2010, ISBN 026201243X, 9780262012430.

[22] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006, ISBN 0387310738.

[23] Webb, A. *Statistical Pattern Recognition*. Wiley InterScience electronic collection, Wiley, 2003, ISBN 9780470854785.

[24] Quinlan, J. R. Induction of decision trees. *Machine learning*, volume 1, no. 1, 1986: pp. 81–106.

[25] Quinlan, J. R.; et al. Bagging, boosting, and C4. 5. In *AAAI/IAAI, Vol. 1*, 1996, pp. 725–730.

[26] Lawrence, R. L.; Wright, A. Rule-based classification systems using classification and regression tree (CART) analysis. *Photogrammetric engineering and remote sensing*, volume 67, no. 10, 2001: pp. 1137–1142.

[27] Mitchell, T. M. *Machine Learning*. New York, NY, USA: McGraw-Hill, Inc., first edition, 1997, ISBN 0070428077, 9780070428072.

[28] Wilson, D. R.; Martinez, T. R. Improved heterogeneous distance functions. *Journal of artificial intelligence research*, 1997.

[29] Carrera, E. Multi-platform Python module to parse and work with PE files - pefile. Available from: `https://github.com/erocarrera/pefile`

[30] Kluyver, T.; Ragan-Kelley, B.; et al. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, edited by F. Loizides; B. Schmidt, IOS Press, 2016, pp. 87 – 90. Available from: `http://jupyter.org/`

[31] McKinney, W. pandas: a Foundational Python Library for Data Analysis and Statistics. Available from: `https://pandas.pydata.org/pandas-docs/stable/index.html`

[32] Pedregosa, F.; Varoquaux, G.; et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, volume 12, 2011: pp. 2825–2830. Available from: `http://scikit-learn.org/stable/index.html`

Appendix A

Acronyms

PE Portable Executable - Microsoft Windows' format for executable files and libraries

DLL Format for Microsoft Windows' dynamically loaded library

AV Anti virus software

IoT Internet of Things

SVM Support Vector Machine

MD5 Message Digest algorithm

HVDM Heterogeneous value difference metric

FPR False positive rate

FNR False negative rate

K-NN K-nearest neighbours

Appendix B

Contents of enclosed CD drive

- `src` ..... the directory of source codes
  - `notebooks` ..... the directory of jupyter notebooks
  - `tools` ..... the directory of tools used
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format