MACHINE LEARNING METHODS FOR MALWARE DETECTION AND CLASSIFICATION

Bachelor’s Thesis
Information Technology

2017
Malware detection is an important factor in the security of the computer systems. However, currently utilized signature-based methods cannot provide accurate detection of zero-day attacks and polymorphic viruses. That is why the need for machine learning-based detection arises.

The purpose of this work was to determine the best feature extraction, feature representation, and classification methods that result in the best accuracy when used on the top of Cuckoo Sandbox. Specifically, k-Nearest-Neighbors, Decision Trees, Support Vector Machines, Naive Bayes and Random Forest classifiers were evaluated. The dataset used for this study consisted of the 1156 malware files of 9 families of different types and 984 benign files of various formats.

This work presents recommended methods for machine learning based malware classification and detection, as well as the guidelines for its implementation. Moreover, the study performed can be useful as a base for further research in the field of malware analysis with machine learning methods.

Keywords
malware, machine learning, classification, malware detection, malware analysis, k-Nearest Neighbors, Decision Tree, Support Vector Machines, Random Forest, Naive Bayes
## CONTENTS

1 INTRODUCTION ........................................................................................................ 5

2 THEORETICAL BACKGROUND ............................................................................. 6
  2.1 Malware types .................................................................................................... 6
  2.2 Detection methods .............................................................................................. 8
  2.3 Need for machine learning .................................................................................. 10
  2.4 Related work ....................................................................................................... 11

3 MACHINE LEARNING METHODS ....................................................................... 12
  3.1 Machine Learning Basics .................................................................................. 12
    3.1.1 Feature extraction ....................................................................................... 14
    3.1.2 Supervised and Unsupervised Learning ..................................................... 15
  3.2 Classification methods ....................................................................................... 16
    3.2.1 K-nearest neighbours ............................................................................... 17
    3.2.2 Support Vector Machines .......................................................................... 19
    3.2.3 Naive Bayes ............................................................................................... 21
    3.2.4 J48 Decision Tree ...................................................................................... 22
    3.2.5 Random Forest ............................................................................................ 24
  3.3 Cross-validation .................................................................................................. 26

4 PRACTICAL PART .................................................................................................. 27
  4.1 Data ................................................................................................................... 28
    4.1.1 Dridex .......................................................................................................... 28
    4.1.2 Locky .......................................................................................................... 30
    4.1.3 Teslacrypt .................................................................................................... 32
    4.1.4 Vawtrak ...................................................................................................... 34
    4.1.5 Zeus ............................................................................................................ 36
    4.1.6 DarkComet ................................................................................................... 37
    4.1.7 CyberGate .................................................................................................... 38
    4.1.8 Xtreme ......................................................................................................... 39
    4.1.9 CTB-Locker ................................................................................................. 40
  4.2 Cuckoo Sandbox ................................................................................................ 41
    4.2.1 Scoring system ............................................................................................. 44
    4.2.2 Reports and features ................................................................................... 46
  4.3 Feature representation ....................................................................................... 48
    4.3.1 Binary representation .................................................................................. 49
    4.3.2 Frequency representation ............................................................................ 49
4.3.3 Combining representation .......................................................... 50
4.4 Feature selection ............................................................................. 50
4.5 Implementation ................................................................................ 51
4.5.1 Sandbox configuration ................................................................. 52
4.5.2 Feature extraction ........................................................................ 52
4.5.3 Feature selection ........................................................................... 54
4.5.4 Application of machine learning methods .................................... 55
5 RESULTS AND DISCUSSION ................................................................. 56
5.1 K-Nearest Neighbors ...................................................................... 56
5.2 Support Vector Machines ................................................................. 59
5.3 J48 Decision Tree ............................................................................ 62
5.4 Naive Bayes .................................................................................... 67
5.5 Random Forest ................................................................................ 69
6 CONCLUSIONS .................................................................................... 72
6.1 Future Work ..................................................................................... 73
BIBLIOGRAPHY ...................................................................................... 75
APPENDICES ............................................................................................. 80
1. Feature Extraction Code (python) ...................................................... 80
2. Feature selection code (R) ............................................................... 85
3. Classification code (R) ..................................................................... 86
4. List of MD5 hashes of malware samples .......................................... 89
1 INTRODUCTION

With the rapid development of the Internet, malware became one of the major cyber threats nowadays. Any software performing malicious actions, including information stealing, espionage, etc. can be referred to as malware. Kaspersky Labs (2017) define malware as “a type of computer program designed to infect a legitimate user's computer and inflict harm on it in multiple ways.”

While the diversity of malware is increasing, anti-virus scanners cannot fulfill the needs of protection, resulting in millions of hosts being attacked. According to Kaspersky Labs (2016), 6,563,145 different hosts were attacked, and 4,000,000 unique malware objects were detected in 2015. In turn, Juniper Research (2016) predicts the cost of data breaches to increase to $2.1 trillion globally by 2019.

In addition to that, there is a decrease in the skill level that is required for malware development, due to the high availability of attacking tools on the Internet nowadays. High availability of anti-detection techniques, as well as ability to buy malware on the black market result in the opportunity to become an attacker for anyone, not depending on the skill level. Current studies show that more and more attacks are being issued by script-kiddies or are automated. (Aliyev 2010).

Therefore, malware protection of computer systems is one of the most important cybersecurity tasks for single users and businesses, since even a single attack can result in compromised data and sufficient losses. Massive losses and frequent attacks dictate the need for accurate and timely detection methods. Current static and dynamic methods do not provide efficient detection, especially when dealing with zero-day attacks. For this reason, machine learning-based techniques can be used. This paper discusses the main points and concerns of machine learning-based malware detection, as well as looks for the best feature representation and classification methods.

The goal of this project is to develop the proof of concept for the machine learning based malware classification based on Cuckoo Sandbox. This sandbox will be utilized for the extraction of the behavior of the malware samples, which will be used as an input to the machine learning algorithms. The goal is to
determine the best feature representation method and how the features should be extracted, the most accurate algorithm that can distinguish the malware families with the lowest error rate.

The accuracy will be measured both for the case of detection of whether the file is malicious and for the case of classification of the file to the malware family. The accuracy of the obtained results will also be assessed in relation to current scoring implemented in Cuckoo Sandbox, and the decision of which method performs better will be made. The study conducted will allow building an additional detection module to Cuckoo Sandbox. However, the implementation of this module is beyond the scope of this project and will not be discussed in this paper.

2 THEORETICAL BACKGROUND

This chapter provides the background that is essential to understand the malware detection and the need for machine learning methods. The malware types relevant to the study are described first, followed by the standard malware detection methods. After that, based on the knowledge gained, the need for machine learning is discussed, along with the relevant work performed in this field.

2.1 Malware types

To have a better understanding of the methods and logic behind the malware, it is useful to classify it. Malware can be divided into several classes depending on its purpose. The classes are as follows:

- **Virus.** This is the simplest form of software. It is simply any piece of software that is loaded and launched without user’s permission while reproducing itself or infecting (modifying) other software (Horton and Seberry 1997).

- **Worm.** This malware type is very similar to the virus. The difference is that worm can spread over the network and replicate to other machines (Smith, et al. 2009).
• **Trojan.** This malware class is used to define the malware types that aim to appear as legitimate software. Because of this, the general spreading vector utilized in this class is social engineering, i.e. making people think that they are downloading the legitimate software (Moffie, et al. 2006).

• **Adware.** The only purpose of this malware type is displaying advertisements on the computer. Often adware can be seen as a subclass of spyware and it will very unlikely lead to dramatic results.

• **Spyware.** As it implies from the name, the malware that performs espionage can be referred to as spyware. Typical actions of spyware include tracking search history to send personalized advertisements, tracking activities to sell them to the third parties subsequently (Chien 2005).

• **Rootkit.** Its functionality enables the attacker to access the data with higher permissions than is allowed. For example, it can be used to give an unauthorized user administrative access. Rootkits always hide its existence and quite often are unnoticeable on the system, making the detection and therefore removal incredibly hard. (Chuvakin 2003).

• **Backdoor.** The backdoor is a type of malware that provides an additional secret “entrance” to the system for attackers. By itself, it does not cause any harm but provides attackers with broader attack surface. Because of this, backdoors are never used independently. Usually, they are preceding malware attacks of other types.

• **Keylogger.** The idea behind this malware class is to log all the keys pressed by the user, and, therefore, store all data, including passwords, bank card numbers and other sensitive information (Lopez, et al. 2013).

• **Ransomware.** This type of malware aims to encrypt all the data on the machine and ask a victim to transfer some money to get the decryption key. Usually, a machine infected by ransomware is “frozen” as the user cannot open any file, and the desktop picture is used to provide information on attacker’s demands. (Savage, Coogan and Lau 2015).
• **Remote Administration Tools (RAT).** This malware type allows an attacker to gain access to the system and make possible modifications as if it was accessed physically. Intuitively, it can be described in the example of the TeamViewer, but with malicious intentions.

### 2.2 Detection methods

All malware detection techniques can be divided into signature-based and behavior-based methods. Before going into these methods, it is essential to understand the basics of two malware analysis approaches: static and dynamic malware analysis. As it implies from the name, static analysis is performed “statically”, i.e. without execution of the file. In contrast, dynamic analysis is conducted on the file while it is being executed for example in the virtual machine.

**Static analysis** can be viewed as “reading” the source code of the malware and trying to infer the behavioral properties of the file. Static analysis can include various techniques (Prasad, Annangi and Pendyala 2016) :

1. **File Format Inspection:** file metadata can provide useful information. For example, Windows PE (portable executable) files can provide much information on compile time, imported and exported functions, etc.

2. **String Extraction:** this refers to the examination of the software output (e.g. status or error messages) and inferring information about the malware operation.

3. **Fingerprinting:** this includes cryptographic hash computation, finding the environmental artifacts, such as hardcoded username, filename, registry strings.

4. **AV scanning:** if the inspected file is a well-known malware, most likely all anti-virus scanners will be able to detect it. Although it might seem irrelevant, this way of detection is often used by AV vendors or sandboxes to “confirm” their results.
5. **Disassembly**: this refers to reversing the machine code to assembly language and inferring the software logic and intentions. This is the most common and reliable method of static analysis.

Static analysis often relies on certain tools. Beyond the simple analysis, they can provide information on protection techniques used by malware. The main advantage of static analysis is the ability to discover all possible behavioral scenarios. Researching the code itself allows the researcher to see all ways of malware execution, that are not limited to the current situation. Moreover, this kind of analysis is safer than dynamic, since the file is not executed and it cannot result in bad consequences for the system. On the other hand, static analysis is much more time-consuming. Because of these reasons it is not usually used in real-world dynamic environments, such as anti-virus systems, but is often used for research purposes, e.g. when developing signatures for zero-day malware. (Prasad, Annangi and Pendyala 2016).

Another analysis type is **dynamic analysis**. Unlike static analysis, here the behavior of the file is monitored while it is executing and the properties and intentions of the file are inferred from that information. Usually, the file is run in the virtual environment, for example in the sandbox. During this kind of analysis, it is possible to find all behavioral attributes, such as opened files, created mutexes, etc. Moreover, it is much faster than static analysis. On the other hand, the static analysis only shows the behavioral scenario relevant to the current system properties. For example, if our virtual machine has Windows 7 installed, the results might be different from the malware running under Windows 8.1. (Egele, et al. 2012).

Now, having the background on malware analysis, we can define the detection methods. The **signature-based analysis** is a static method that relies on pre-defined signatures. These can be file fingerprints, e.g. MD5 or SHA1 hashes, static strings, file metadata. The scenario of detection, in this case, would be as follows: when a file arrives at the system, it is statically analyzed by the anti-virus software. If any of the signatures is matched, an alert is triggered, stating that this file is suspicious. Very often this kind of analysis is enough since well-known malware samples can often be detected based on hash values.
However, attackers started to develop malware in a way that it can change its signature. This malware feature is referred to as polymorphism. Obviously, such malware cannot be detected using purely signature-based detection techniques. Moreover, new malware types cannot be detected using signatures, until the signatures are created. Therefore, AV vendors had to come up with another way of detection – behavior-based also referred to as **heuristics-based analysis**. In this method, the actual behavior of malware is observed during its execution, looking for the signs of malicious behavior: modifying host files, registry keys, establishing suspicious connections. By itself, each of these actions cannot be a reasonable sign of malware, but their combination can raise the level of suspiciousness of the file. There is some threshold level of suspiciousness defined, and any malware exceeding this level raises an alert. (Harley and Lee 2009).

The accuracy level of heuristics-based detection highly depends on the implementation. The best ones utilize the virtual environment, e.g. the sandbox to run the file and monitor its behavior. Although this method is more time-consuming, it is much safer, since the file is checked before actually executing.

The main advantage of behavior-based detection method is that in theory, it can identify not only known malware families but also zero-day attacks and polymorphic viruses. However, in practice, taking into account the high spreading rate of malware, such analysis cannot be considered effective against new or polymorphic malware.

### 2.3 Need for machine learning

As stated before, malware detectors that are based on signatures can perform well on previously-known malware, that was already discovered by some anti-virus vendors. However, it is unable to detect polymorphic malware, that has an ability to change its signatures, as well as new malware, for which signatures have not been created yet. In turn, the accuracy of heuristics-based detectors is not always sufficient for adequate detection, resulting in a lot of false-positives and false-negatives. (Baskaran and Ralescu 2016).

Need for the new detection methods is dictated by the high spreading rate of polymorphic viruses. One of the solutions to this problem is reliance on the
heuristics-based analysis in combination with machine learning methods that offer a higher efficiency during detection.

When relying on heuristics-based approach, there has to be a certain threshold for malware triggers, defining the amount of heuristics needed for the software to be called malicious. For example, we can define a set of suspicious features, such as “registry key changed”, “connection established”, “permission changed”, etc. Then we can state, that any software, that triggers at least five features from that set can be called malicious. Although this approach provides some level of effectiveness, it is not always accurate, since some features can have more “weight” than others, for example, “permission changed” usually results in more severe impact to the system than “registry key changed”. In addition to that, some feature combinations might be more suspicious than features by themselves. (Rieck, et al. 2011).

To take these correlations into account and provide more accurate detection, machine learning methods can be used.

2.4 Related work

Although not widely implemented, the concept of machine learning methods for malware detection is not new. Several types of studies were carried out in this field, aiming to figure the accuracy of different methods.

In his paper “Malware Detection Using Machine Learning” Dragos Gavrilut aimed for developing a detection system based on several modified perceptron algorithms. For different algorithms, he achieved the accuracy of 69.90%-96.18%. It should be stated that the algorithms that resulted in best accuracy also produced the highest number of false-positives: the most accurate one resulted in 48 false positives. The most "balanced"s algorithm with appropriate accuracy and the low false-positive rate had the accuracy of 93.01%. (Gavrilut, et al. 2009).

The paper “Malware Detection Module using Machine Learning Algorithms to Assist in Centralized Security in Enterprise Networks” discusses the detection method based on modified Random Forest algorithm in combination with Information Gain for better feature representation. It should be noticed, that the data set consists purely of portable executable files, for which feature extraction
is generally easier. The result achieved is the accuracy of 97% and 0.03 false-positive rate. (Singhal and Raul 2015).

“A Static Malware Detection System Using Data Mining Methods” proposed extraction methods based on PE headers, DLLs and API functions and methods based on Naive Bayes, J48 Decision Trees, and Support Vector Machines. Highest overall accuracy was achieved with the J48 algorithm (99% with PE header feature type and hybrid PE header&API function feature type, 99.1% with API function feature type). (Baldangombo, Jambaljav and Horng 2013).

In “Zero-day Malware Detection based on Supervised Learning Algorithms of API call Signatures”, the API functions were used for feature representation again. The best result was achieved with Support Vector Machines algorithm with normalized polykernel. The precision of 97.6% was achieved, with a false-positive rate of 0.025. (Alazab, et al. 2011).

As it can be seen, all studies ended up with different results. From here, we can conclude that no unified methodology was created yet neither for detection nor feature representation. The accuracy of each separate case depends on the specifics of malware families used and on the actual implementation.

3 MACHINE LEARNING METHODS

This chapter gives a theoretical background on machine learning methods, needed for understanding the practical implementation. First, the overview of the machine learning field is discussed, followed by the description of methods relevant to this study. These methods include k-Nearest Neighbors, Decision Trees, Random Forests, Support Vector Machines and Naive Bayes.

3.1 Machine Learning Basics

The rapid development of data mining techniques and methods resulted in Machine Learning forming a separate field of Computer Science. It can be viewed as a subclass of the Artificial Intelligence field, where the main idea is the ability of a system (computer program, algorithm, etc.) to learn from its own actions. It was firstly referred to as "field of study that gives computers the ability to learn without being explicitly programmed" by Arthur Samuel in 1959. A more formal definition is given by T. Mitchell: "A computer program is said to learn
from experience $E$ with respect to some class of tasks $T$ and performance measure $P$ if its performance at tasks in $T$, as measured by $P$, improves with experience $E$." (Mitchell 1997).

The basic idea of any machine learning task is to train the model, based on some algorithm, to perform a certain task: classification, clusterization, regression, etc. Training is done based on the input dataset, and the model that is built is subsequently used to make predictions. The output of such model depends on the initial task and the implementation. Possible applications are: given data about house attributes, such as room number, size, and price, predict the price of the previously unknown house; based on two datasets with healthy medical images and the ones with tumor, classify a pool of new images; cluster pictures of animals to several clusters from an unsorted pool.

To develop a deeper understanding, it is worth going through the general workflow of the machine learning process, which is shown in Figure 1.

As it can be seen, the process consists of 5 stages:

1. **Data intake.** At first, the dataset is loaded from the file and is saved in memory.

2. **Data transformation.** At this point, the data that was loaded at step 1 is transformed, cleared, and normalized to be suitable for the algorithm. Data is converted so that it lies in the same range, has the same format, etc. At this point feature extraction and selection, which are discussed further, are performed as well. In addition to that, the data is separated into sets – ‘training set’ and ‘test set’. Data from the training set is used to build the model, which is later evaluated using the test set.
3. **Model Training.** At this stage, a model is built using the selected algorithm.

4. **Model Testing.** The model that was built or trained during step 3 is tested using the test data set, and the produced result is used for building a new model, that would consider previous models, i.e. “learn” from them.

5. **Model Deployment.** At this stage, the best model is selected (either after the defined number of iteration or as soon as the needed result is achieved).

3.1.1 **Feature extraction**

In any of the examples mentioned above, we should be able to extract the attributes from the input data, so that it can be fed to the algorithm. For example, for the housing prices case, data could be represented as a multidimensional matrix, where each column represents an attribute and rows represent the numerical values for these attributes. In the image case, data can be represented as an RGB value of each pixel.

Such attributes are referred to as *features*, and the matrix is referred to as feature vector. The process of extracting data from the files is called feature extraction. The goal of feature extraction is to obtain a set of informative and non-redundant data. It is essential to understand that features should represent the important and relevant information about our dataset since without it we cannot make an accurate prediction. That is why feature extraction is often a non-obvious task, which requires a lot of testing and research. Moreover, it is very domain-specific, so general methods apply here poorly.

Another important requirement for a decent feature set is non-redundancy. Having redundant features i.e. features that outline the same information, as well as redundant information attributes, that are closely dependent on each other, can make the algorithm biased and, therefore, provide an inaccurate result.

In addition to that, if the input data is too big to be fed into the algorithm (has too many features), then it can be transformed to a reduced feature vector
(vector, having a smaller number of features). The process of reducing the vector dimensions is referred to as feature selection. At the end of this process, we expect the selected features to outline the relevant information from the initial set so that it can be used instead of initial data without any accuracy loss.

Other possible transformations are:

1. **Normalization**
   
   An example of normalization can be dividing an image $x_i$ where $x_i$s are the number of pixels with color $i$, by the total number of counts to encode the distribution and remove the dependence on the size of the image. This translates into the formula: $x' = \frac{x}{||x||}$ (Guyon and Elisseef 2006).

2. **Standardization**
   
   Sometimes, even while referring to comparable objects, features can have different scales. For example, consider the housing prices example. Here, feature ‘room size’ is an integer, probably not exceeding 5 and feature ‘house size’ is measured in square meters. Although both values can be compared, added, multiplied, etc., the result would be unreasonable before normalization. The following scaling is often done:

   $$x'_i = (x_i - \mu_i)/\sigma_i$$, where $\mu_i$ and $\sigma_i$ are the mean and the standard deviation of feature $x_i$ over training examples. (Guyon and Elisseef 2006).

3. **Non-linear expansions**
   
   Although in most cases we want to reduce the dimensionality of data, in some cases it might make sense to increase it. This can be useful for complex problems, where first-order interactions are not sufficient for accurate results.

3.1.2 **Supervised and Unsupervised Learning**

So far we have discussed the machine learning concepts from the point of view, where we have initial data, on which the model can be trained. However, this is not always the case. Here we want to introduce the two machine learning approaches - supervised and unsupervised learning.
In **Supervised Learning**, learning is based on labeled data. In this case, we have an initial dataset, where data samples are mapped to the correct outcome. The housing prices case is an example of supervised learning: here we have an initial dataset with houses, its attributes, and its prices. The model is trained on this dataset, where it "knows" the correct results. Examples of supervised learning are regression and classification problems:

1. **Regression**
   Predict the value based on previous observations, i.e. values of the samples from the training set. Usually, we can say that if the output is a real number/is continuous, then it is a regression problem.

2. **Classification**
   Based on the set of labeled data, where each label defines a class, that the sample belongs to, we want to predict the class for the previously unknown sample. The set of possible outputs is finite and usually small. Generally, we can say that if the output is a discrete/categorical variable, then it is a classification problem.

In contrast to Supervised Learning, in **Unsupervised Learning**, there is no initial labeling of data. Here the goal is to find some pattern in the set of unsorted data, instead of predicting some value. A common subclass of Unsupervised Learning is Clustering:

3. **Clustering**
   Find the hidden patterns in the unlabeled data and separate it into clusters according to similarity. An example can be the discovery of different customer groups inside the customer base of the online shop.

### 3.2 Classification methods

From machine learning perspective, malware detection can be seen as a problem of classification or clusterization: unknown malware types should be clusterized into several clusters, based on certain properties, identified by the algorithm. On the other hand, having trained a model on the wide dataset of malicious and benign files, we can reduce this problem to classification. For known malware families, this problem can be narrowed down to classification only – having a limited set of classes, to one of which malware sample certainly
belongs, it is easier to identify the proper class, and the result would be more accurate than with clusterization algorithms. In this section, the theoretical background is given on all the methods used in this project.

3.2.1 K-nearest neighbours

K-Nearest Neighbors (KNN) is one of the simplest, though, accurate machine learning algorithms. KNN is a non-parametric algorithm, meaning that it does not make any assumptions about the data structure. In real world problems, data rarely obeys the general theoretical assumptions, making non-parametric algorithms a good solution for such problems. KNN model representation is as simple as the dataset – there is no learning required, the entire training set is stored.

KNN can be used for both classification and regression problems. In both problems, the prediction is based on the $k$ training instances that are closest to the input instance. In the KNN classification problem, the output would be a class, to which the input instance belongs, predicted by the majority vote of the $k$ closest neighbors. In the regression problem, the output would be the property value, which is generally a mean value of the $k$ nearest neighbors. The schematic example is outlined in Figure 2.

![Figure 2. KNN example](image)

Different distance measurement methods are used for finding the closest neighbors. The popular ones include Hamming Distance, Manhattan Distance, Minkowski distance:
Hamming Distance: $d_{ij} = \sum_{k=1}^{p} |x_{ik} - x_{jk}|$  \[1\]

Manhattan Distance: $d_1(p, q) = ||p - q||_1 = \sum_{i=1}^{n} |p_i - q_i|$  \[2\]

Minkowski Distance = $\left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p}$  \[3\]

The most used method for continuous variables is generally the **Euclidean Distance**, which is defined by the formulae below:

$$ EuclidianDistance = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} ; p and q are the points in n - space $$  \[4\]

Euclidian distance is good for the problems, where the features are of the same type. For the features of different types, it is advised to use, for example, Manhattan Distance.

For the classification problems, the output can also be presented as a set of probabilities of an instance belonging to the class. For example, for binary problems, the probabilities can be calculated like $P(0) = \frac{N_0}{N_0 + N_1}$, where $P(0)$ is the probability of the 0 class membership and $N_0, N_1$ are numbers of neighbors belonging to the classes 0 and 1 respectively. (Thirumuruganathan 2010).

The value of $k$ plays a crucial role in the prediction accuracy of the algorithm. However, selecting the $k$ value is a non-trivial task. Smaller values of $k$ will most likely result in lower accuracy, especially in the datasets with much noise, since every instance of the training set now has a higher weight during the decision process. Larger values of $k$ lower the performance of the algorithm. In addition to that, if the value is too high, the model can overfit, making the class boundaries less distinct and resulting in lower accuracy again. As a general approach, it is advised to select $k$ using the formula below:

$$ k = \sqrt{n} $$  \[5\]
For classification problems with an even number of classes, it is advised to choose an odd $k$ since this will eliminate the possibility of a tie during the majority vote.

The drawback of the KNN algorithm is the bad performance on the unevenly distributed datasets. Thus, if one class vastly dominates the other ones, it is more likely to have more neighbors of that class due to their large number, and, therefore, make incorrect predictions. (Laaksonen and Oja 1996).

### 3.2.2 Support Vector Machines

Support Vector Machines (SVM) is another machine learning algorithm that is generally used for classification problems. The main idea relies on finding such a hyperplane, that would separate the classes in the best way. The term ‘support vectors’ refers to the points lying closest to the hyperplane, that would change the hyperplane position if removed. The distance between the support vector and the hyperplane is referred to as margin.

Intuitively, we understand that the further from the hyperplane our classes lie, the more accurate predictions we can make. That is why, although multiple hyperplanes can be found per problem, the goal of the SVM algorithm is to find such a hyperplane that would result in the maximum margins.

![SVM scheme](image.png)
On Figure 3, there is a dataset of two classes. Therefore, the problem lies in a two-dimensional space, and a hyperplane is represented as a line. In general, hyperplane can take as many dimensions as we want.

The algorithm can be described as follows:

1. We define $X$ and $Y$ as the input and output sets respectively. $(x_1, y_1),\ldots,(x_m, y_m)$ is the training set.
2. Given $x$, we want to be able to predict $y$. We can refer to this problem as to learning the classifier $y=f(x,a)$, where $a$ is the parameter of the classification function.
3. $F(x,a)$ can be learned by minimizing the training error of the function that learns on training data. Here, $L$ is the loss function, and $R_{emp}$ is referred to as empirical risk.

$$R_{emp}(a) = \frac{1}{m} \sum_{i=1}^{m} l(f(x_i,a), y_i) = \text{Training Error} \quad [6]$$

4. We are aiming at minimizing the overall risk, too. Here, $P(x,y)$ is the joint distribution function of $x$ and $y$.

$$R(a) = \int l(f(x,a), y) dP(x,y) = \text{Test Error} \quad [7]$$

5. We want to minimize the $\text{Training Error + Complexity term}$. So, we choose the set of hyperplanes, so $f(x) = (w \cdot x) + b$:

$$\frac{1}{m} \sum_{i=1}^{m} l(w \cdot x_i + b, y_i) + \|w\|^2 \text{ subject to } \min_i |w \cdot x_i| = 1 \quad [8]$$

SVMs are generally able to result in good accuracy, especially on "clean" datasets. Moreover, it is good with working with the high-dimensional datasets, also when the number of dimensions is higher than the number of the samples. However, for large datasets with a lot of noise or overlapping classes, it can be more effective. Also, with larger datasets training time can be high. (Jing and Zhang 2010).
3.2.3 Naive Bayes

Naive Bayes is the classification machine learning algorithm that relies on the Bayes Theorem. It can be used for both binary and multi-class classification problems. The main point relies on the idea of treating each feature independently. Naive Bayes method evaluates the probability of each feature independently, regardless of any correlations, and makes the prediction based on the Bayes Theorem. That is why this method is called "naive" – in real-world problems features often have some level of correlation between each other.

To understand the algorithm of Naive Bayes, the concepts of class probabilities and conditional probabilities should be introduced first.

   a. **Class Probability** is a probability of a class in the dataset. In other words, if we select a random item from the dataset, this is the probability of it belonging to a certain class.

   b. **Conditional Probability** is the probability of the feature value given the class.

1. Class probability is calculated simply as the number of samples in the class divided by the total number of samples:

\[
P(C) = \frac{\text{count}(\text{instances in } C)}{\text{count}(\text{instances in Ntotal})}\]  \[9\]

2. Conditional probabilities are calculated as the frequency of each attribute value divided by the frequency of instances of that class.

\[
P(V|C) = \frac{\text{count}(\text{instances with } V \text{ and } C)}{\text{count}(\text{instances with } V)}\]  \[10\]

3. Given the probabilities, we can calculate the probability of the instance belonging to a class and therefore make decisions using the Bayes Theorem:

\[
P(A|B) = \frac{P(B|A)P(A)}{P(B)}\]  \[11\]
4. Probabilities of the item belonging to all classes are compared and the class with the highest probability if selected as a result.

The advantages of using this method include its simplicity and easiness of understanding. In addition to that, it performs well on the data sets with irrelevant features, since the probabilities of them contributing to the output are low. Therefore they are not taken into account when making predictions. Moreover, this algorithm usually results in a good performance in terms of consumed resources, since it only needs to calculate the probabilities of the features and classes, there is no need to find any coefficients like in other algorithms. As already mentioned, its main drawback is that each feature is treated independently, although in most cases this cannot be true. (Bishop 2006).

3.2.4 J48 Decision Tree

As it implies from the name, decision trees are data structures that have a structure of the tree. The training dataset is used for the creation of the tree, that is subsequently used for making predictions on the test data. In this algorithm, the goal is to achieve the most accurate result with the least number of the decisions that must be made. Decision trees can be used for both classification and regression problems. An example can be seen in Table 1:

<table>
<thead>
<tr>
<th>Outlook</th>
<th>Temperature</th>
<th>Humidity</th>
<th>Windy</th>
<th>Play tennis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rainy</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>Rainy</td>
<td>Hot</td>
<td>High</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>Overcast</td>
<td>Hot</td>
<td>High</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>Sunny</td>
<td>Mild</td>
<td>High</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>Sunny</td>
<td>Cool</td>
<td>Normal</td>
<td>False</td>
<td>Yes</td>
</tr>
<tr>
<td>Overcast</td>
<td>Cool</td>
<td>High</td>
<td>True</td>
<td>No</td>
</tr>
<tr>
<td>Rainy</td>
<td>Cool</td>
<td>High</td>
<td>True</td>
<td>Yes</td>
</tr>
<tr>
<td>Rainy</td>
<td>Mild</td>
<td>High</td>
<td>False</td>
<td>No</td>
</tr>
<tr>
<td>Sunny</td>
<td>Cool</td>
<td>Normal</td>
<td>False</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1. Decision tree example dataset
As it can be seen in Figure 4, the model was trained based on the dataset and can now classify the tennis playing decision to “yes” or “no”. Here, the tree consists of the decision nodes and leaf nodes. Decision nodes have several branches leading to leaf nodes. Leaf nodes represent the decisions or classifications. The topmost initial node is referred to as root node.

The common algorithm for decision trees is **ID3 (Iterative Dichotomiser 3)**. It relies on the concepts of the **Entropy** and **Information Gain**. Entropy here refers to the level of uncertainty in the data content. For example, the entropy of the coin toss would be indefinite, since there is no way to be sure in the result. Contrarily, a coin toss of the coin with two heads on both sides would result in zero entropy, since we can predict the outcome with 100% probability before each toss. (Mitchell 1997).

In simple words, the ID3 algorithm can be described as follows: starting from the root node, at each stage we want to partition the data into homogenous (similar in their structure) dataset. More specifically, we want to find the attribute that would result in the highest information gain, i.e. return the most homogenous branches (Swain and Hauska 1977):

1. Calculate the entropy of the target.

\[
E(T, X) = \sum_{c \in X} P(c)E(c) \tag{12}
\]

\[
E(S) = \sum_{i=1}^{c} -p_i \log_2 p_i \tag{13}
\]
2. Split the dataset and calculate the entropy of each branch. Then calculate the information gain of the split, that is the differences in the initial entropy and the proportional sum of the entropies of the branches.

\[ Gain(T, X) = Entropy(T) - Entropy(T, X) \]  

3. The attribute with the highest Gain value is selected as the decision node.

4. If one of the branches of the selected decision node has an entropy of 0, it becomes the leaf node. Other branches require further splitting.

5. The algorithm is run recursively until there is nothing to split anymore.

J48 is the implementation of the ID3 algorithm, that is included in one of the R packages, and this is the implementation we are going to use in our study.

Decision tree method achieved its popularity because of its simplicity. It can deal well with large datasets and can handle the noise in the datasets very well. Another advantage is that unlike other algorithms, such as SVM or KNN, decision trees operate in a “white box”, meaning that we can clearly see how the outcome is obtained and which decisions led to it. These facts made it a popular solution for medical diagnosis, spam filtering, security screening and other fields. (Mitchell 1997).

### 3.2.5 Random Forest

Random Forest is one of the most popular machine learning algorithms. It requires almost no data preparation and modeling but usually results in accurate results. Random Forests are based on the decision trees described in the previous section. More specifically, Random Forests are the collections of decision trees, producing a better prediction accuracy. That is why it is called a 'forest' – it is basically a set of decision trees.

The basic idea is to grow multiple decision trees based on the independent subsets of the dataset. At each node, n variables out of the feature set are selected randomly, and the best split on these variables is found.

In simple words, the algorithm can be described as follows (Biau 2013):
1. Multiple trees are built roughly on the two third of the training data (62.3%). Data is chosen randomly.

2. Several predictor variables are randomly selected out of all the predictor variables. Then, the best split on these selected variables is used to split the node. By default, the amount of the selected variables is the square root of the total number of all predictors for classification, and it is constant for all trees.

3. Using the rest of the data, the misclassification rate is calculated. The total error rate is calculated as the overall out-of-bag error rate.

4. Each trained tree gives its own classification result, giving its own "vote". The class that received the most "votes" is chosen as the result. The scheme of the algorithm is seen in Figure 5.

As in the decision trees, this algorithm removes the need for feature selection for removing irrelevant features – they will not be taken into account in any case. The only need for any feature selection with the random forest algorithms arises
when there is a need for dimensionality reduction. Moreover, the out-of-bag error rate, which was mentioned earlier can be considered the algorithm’s own cross-validation method. This removes the need for tedious cross-validation measures, that would have to be taken otherwise. (Mitchell 1997).

Random forests inherit many of the advantages of the decision trees algorithms. They are applicable to both regression and classification problems; they are easy to compute and quick to fit. They also usually result in the better accuracy. However, unlike decision trees, it is not very easy to interpret the results. In decision trees, by examining the resulting tree, we can gain valuable information about which variables are important and how they affect the result. This is not possible with random forests. It can also be described as a more stable algorithm than the decision trees – if we modify the data a little bit, decision trees will change, most likely reducing the accuracy. This will not happen in the random forest algorithms – since it is the combination of many decision trees, the random forest will remain stable. (Louppe 2014).

### 3.3 Cross-validation

The drawback of the accuracy evaluation methods that are present in the machine learning methods themselves is that they cannot predict how the model will perform on the new data. The approach to overcoming this drawback relies on the cross-validation. The idea is to split the initial dataset. The model is trained on the biggest part of the dataset and then subsequently tested on the smaller part. There are three different classes of cross-validation:

1. **Holdout method** – here, the dataset is separated into two parts: a training set and test set. The model is fit on the training set. The model is then tested on the test set, which it has not seen before. The resulting errors are used to compute the mean absolute test error, that is used for model evaluation. The advantage of this method is its high speed. On the other hand, the evaluation result depends highly on how the test set was selected since the variance is usually high. Therefore, the evaluation result can differ significantly between different test sets.
2. The k-fold method can be seen as the improvement over the holdout method. Here, the $k$ subsets are selected, and the holdout method is repeated $k$ times, where each time one of the $k$ subsets is used as a training set, and the $k-1$ subset is used as the test set. The average error is then computed over all $k$ runs of holdout method. With the increase of $k$, the variance is reduced, ensuring that the accuracy will not change with different datasets. The disadvantage is the complexity and the running time, which is higher as compared to the holdout method.

3. The leave-one-out method is the extreme case of the k-fold method, where the $k$ is as big as the sample universe. On each run of the holdout method, data is trained on all the data points except from one, and that one point is subsequently used for testing. The variance, in this case, is as small as possible. The computing complexity, on the other hand, is high. (Schneider 1997).

This chapter provided background on the machine learning that is essential for understanding the practical implementation of the project, that is described in the next chapter. The concepts of feature set, feature extraction, and selection methods were discussed along with the machine learning algorithms that will be used in practical part. The chosen algorithms are K-Nearest Neighbours, Support Vector Machines, Decision Trees, Random Forests and Naive Bayes.

4 PRACTICAL PART

As a reminder, the goal of the project lies in the determination of the most suitable feature representation and extraction methods, the most accurate algorithm that can distinguish the malware families with the lowest error rate and how this accuracy relates to the current scoring system accuracy. This chapter discusses the practical aspects of the project implementation. This includes data gathering, description of malware families that represent the dataset, selection of the features that will be used for the algorithm and finding the optimal feature representation method, evaluation method, and the implementation process.
4.1 Data

For this project, a total of 2,140 files were collected. For most of them, hashes, which uniquely identify files were found in incidence reports or malware reverse engineering reports, and these hashes were subsequently used to get the corresponding samples from the VirusTotal service with the help of external malware researchers. (VirusTotal 2017) To be able to operate with a diverse dataset, nine malware families were used, resulting in 1,156 malicious files and 984 benign files. Malicious families that were used are Dridex, Locky, TeslaCrypt, Vawtrak, Zeus, DarkComet, CyberGate, Xtreme, CTB-Locker. They are discussed in detail further in this chapter. Benign files were mainly software installers of the .exe format, but also included several files of .pdf, .docx, etc. formats, as they are often used as malware spreading vectors. To achieve the most meaningful and up-to-date results only malware that has appeared in the last two years is used.

4.1.1 Dridex

The first malware family with a total of 172 unique files is Dridex. This malware belongs to the Trojan class, specifically, banking trojan. It caused a huge infection in 2015, resulting in 3,000 - 5,000 infections per month.

Dridex is derived from Cridex, malware that spread in 2012. Cridex was also a banking credentials stealer, but more specifically, it was a worm, that utilized attached storage devices as a spreading vector. In 2014 a renewed version appeared, switching from command and control communications to peer-to-peer and therefore becoming more resilient to takedown operations.

The Dridex attack was targeted to users of specific banks, aiming to steal their credentials during banking sessions. It is said to be target over 300 institutions and 40 regions, mostly focusing on English-speaking countries with high income rates: most infections happened in the United States, the United Kingdom, and Australia. (O’Brien 2016).

Most of the Dridex malware files were distributed during a massive-scale spam campaign, by using real company names as the sender addresses, but fake top level domains, matching the location of the targeted users. Most emails were either invoices or orders. Attackers behind Dridex showed a high level of
attention to details: emails with real company names also utilized real employee names and were sent during business hours.

![Dridex Operation Scheme Diagram](image)

Figure 6. Dridex operation scheme (Aquino 2014)

The operation scheme of Dridex is outlined in Figure 6. From a technical perspective, Dridex malware was embedded into macros of Word documents. After running these macros, a file of .vbs format was run and executed, downloading and installing Dridex Trojan. Dridex performs a man-in-the-middle attack, embedding itself into the Chrome, Firefox or Internet Explorer web browsers and subsequently monitoring traffic and seeking for online banking connections. After finding one, Dridex steals data from keylogs, screenshots, and input forms. (O'Brien 2016).

Dridex has a modular architecture, allowing for the attackers to easily add additional functionality. According to Symantec, there are the following modules (O'Brien 2016):

1. **Loader module** s only purpose is to install the main module. The loader will find one of the servers inside of its configuration and request a binary and configuration data using HTTPS request.
2. **Main module** performs the most functionality of the Dridex malware, including taking screenshots, logging keystrokes, stealing data input forms, deleting files, stealing cookies, etc. For communication, it uses HTTPS with gzipped and XOR-encrypted data.

3. **VNC (Virtual Network Computing) module**, which is available on both x86 and x64 architectures provides a graphical interface for the remote control of the computer. It supports a wide variety of functions, including command prompt, disk management, system settings, etc.

4. **SOCKS module** is also available for x86 and x64 architectures, supporting remote command execution, file download, command and control, etc.

5. **The mod4 module** is used for the creation of new processes.

6. **The mod6 module** provides an ability to send emails via Outlook and is used for spam campaigns.

### 4.1.2 Locky

The second malware family, represented by 115 unique files, is Locky. This is ransomware that encrypts all data on the victim’s system using the RSA-2048 and AES-256 ciphers and adds a .locky extension to it. Locky emerged in February 2016 and has been distributed aggressively since then. The most common distribution vectors are spam campaigns, specifically, fake invoices and phishing websites. These spam campaigns were extremely similar to the ones used to distribute Dridex in its size, utilization of financial documents and macros, which gives a sign of the Dridex group being responsible for this malware. The price for decryption of system files varied from 0.5 to 1 bitcoin. (Symantec Security Response 2016). The operation scheme of Locky can be seen in Figure 7.
Upon delivery to the system, the macros embedded into a .docx or .xls file runs and downloads the Locky malware. Malware file, in turn, injects itself into the %temp% folder with a random name and .exe or .dll file format. A “Run” registry key with value “Locky” will subsequently be added to the registry, pointing out the .exe file in the %temp% folder. The initial file will be deleted at this point. A new process will be started after that, exploring the volume properties and deleting shadow copies present on the volume. Recovery instructions and the public key are retrieved with a POST request from a command and control server. After that, all files on the system are encrypted, and the desktop
background is changed to the image with the decryption instructions. (McAfee Labs 2016). An additional registry key is created, allowing the malware to run every time the system is started. Figure 8 shows the decryption instructions for Locky.

![Image of decryption instructions for Locky malware](image)

Figure 8. Recovery instructions of the Locky malware (Symantec Security Response 2016)

### 4.1.3 Teslacrypt

Teslacrypt is the third malware family, consisting of 115 files and belonging to the ransomware class. Main distribution vector is compromised websites and emails with links leading to malicious websites that download the malware once they are visited. Upon download, the file is executed immediately. The operation scheme of Teslacrypt can be found in Figure 9.
Upon execution TeslaCrypt is copied to the /AppData/Roaming/ folder. Malware is compiled with a C++ compiler and the screen outlined in Figure 10 pops up upon the encryption is finished (McAfee Labs 2016):
Payment for a decryption key is requested to be made via PayPal or Bitcoin (1000 USD or 1.5 bitcoin). Unlike other ransomware families, TeslaCrypt encrypted not only obvious data files, such as .pdf, .doc, .jpg etc., but also game-related files, including Call of Duty, World of Tanks, Minecraft and World of Warcraft.

Interestingly, in May 2016, the attackers behind TeslaCrypt announced that they closed the project and released the master decryption key. Several days later, ESET antivirus released a free decryption tool. More details can be found in Figure 11.

Figure 11. Payment page of TeslaCrypt with the master decryption key (Mimoso 2016)

4.1.4 Vawtrak

Fourth malware family that consists of 74 unique files is Vawtrak. Also referred to as Neverquest or Snifula, Vawtrak is another example of banking Trojan. The most infections happened in Czech Republic, USA, UK, and Germany. Spreading vectors include malware downloaders, spam with malicious links or other drive-by downloads. After downloading, Vawtrak is capable of gaining access to banking accounts of a victim, as well as stealing credentials, passwords, private keys, etc.

The operation process of this malware family is outlined in Figure 12. The execution of the initial file, downloaded to the drive, results in the installation of a dropper file into %ProgramData% folder with a randomly created extension and filename. The initial file is deleted after that. (Křoustek 2015). This dropper file is a DLL that is responsible for unpacking the Vawtrak module and injecting it to the running processes. To do that, the DLL firstly decrypts the payload with
the hardcoded key and decompresses itself, resulting in a new DLL, which replaces the initial one. This DLL, in turn, extracts the final module, which turns out to be a compressed version of two DLLs: 64 and 32-bit modifications. These DLLs are injected into the system processes and are responsible for the Vawtrak’s functionality.

After successful execution, Vawtrak is capable of performing a wide range of malicious actions (Křoustek 2015):

- Disabling the antivirus protection
- Communication with CnC servers
- Stealing passwords, cookies, digital certificates
- Creation of a proxy server on the host system
- Keylogging and screenshots taking
- Changing web browser settings (Internet Explorer, Firefox, Google Chrome) and modifying communications with web servers
4.1.5 Zeus

Zeus is the fifth malware family and is represented by 116 unique files. It is a botnet package, which can be easily traded on the black market for around 700 USD. After its appearance in 2007, Zeus has evolved and remains one of the most common botnet malware representatives.

The summary of Zeus operation can be found in Figure 13. Infection vectors of Zeus vary dramatically, starting from spam emails, and ending with drive-by downloads. After the download, the malware injects itself into the sdra64.exe process and modifies the registry values so that it is executed upon system startup. After that, Zeus injects itself into the winlogon.exe process and terminates the initial executable. Winlogon injected code injects additional data into the svchost.exe process and creates two files: local.ds contains the up-to-date configurations, and user.ds contains data to be transmitted to the command and control server. (Falliere and Chien 2009).
The functionality of Zeus includes stealing of system information, online credentials, storage information. Specification of data to be stolen is either hard coded into the binary or is retrieved from the command and control server. (Falliere and Chien 2009).

The popularity of Zeus malware is related to the fact that it is relatively cheap and easy to use. Moreover, it comes as a ready-to-deploy package and as a result can be used by novices and script kiddies.

### 4.1.6 DarkComet

DarkComet is an example of the Remote Administration Tool (RAT). It was utilized in several attacks in 2012-2015. Initially, DarkComet was not developed as a malicious tool, however, because of its nature and functionality, it was eventually used by the Syrian government for espionage, followed by several other attacks in the following years.

During Syrian conflict in 2014, it was used by the Syrian government for espionage on Syrian citizens that were bypassing government’s censorship on the Internet. In 2015, the "Je Suis Charlie" slogan was used to trick people into downloading the DarkComet: it was disguised as a picture, which compromised the users once downloaded.

As most of the RATs, the DarkComet includes two components: the client and the server. However, they have a reverse meaning from the perspective of the attacker, where the 'server' is the machine with malware, and the 'client' is the attacker. The DarkComet relies on the remote-connection architecture: once it executes, the server connects to the client, which has a GUI, allowing it to control the server. (Kujawa 2012) The functionality of DarkComet is broad, including, but not limited to (Kujawa 2012):

- Webcam and sound capture
- Keylogging
- Power off/Shutdown/Restart
- Remote Desktop functionality
- Active ports discovery
- LAN computers discovery
- URL download
- WiFi Access Points discovery
- Remote Edit Service
- Update server from file or URL
- Lock computer
- Redirect IP/port

The communication between the server and the client is outlined in Figure 14.

![Diagram of DarkComet communication scheme](image)

**Figure 14. DarkComet communication scheme**

### 4.1.7 CyberGate

CyberGate is another example of the Remote Administration Tool (RAT). Written in Delphi, it is constantly being developed, resulting in stability and extensive functionality. It should be mentioned that CyberGate can be considered "legal" malware since it was initially developed for legal purposes and is used in legal problems. However, it is often used for malicious activity, such as espionage.
CyberGate provides the ability to:

- Log into the victim’s machine
- Retrieve the screenshots of the machine
- Connect to the multiple users at the same time.
- Lock computer
- Restart, shutdown
- Read and modify the registry
- Interact via shell
- Capture data from connected input devices

The operation of the CyberGate is guided by the attacker, and the communication happens with a client-server model. Again, here the attacker is referred to as a client and the infected machine is a server. The communication happens in a way similar to the one outlined in Figure 14.

In addition to that, there are plenty of the tutorials that can be found on the Internet, allowing people with a limited set of skills to take advantage of this RAT for malicious purposes. (Aziz 2014).

### 4.1.8 Xtreme

Another example of RAT is Xtreme. Developed in Delphi, it is available for free and shares the source code with several other Delphi RAT malware, including CyberGate.

Xtreme was used in several governmental attacks, as well as several attacks targeting Israel and Palestine. The architecture of Xtreme relies on the client-server architecture, where the attacker is considered to be a client. The configurations are written to the %APPDATA%\Microsoft\Windows folder or the folder named after the mutex created. The data is subsequently encrypted using RC4 and "CONFIG" or "CYBERGATEPASS" as a password. The configurations are stored in the file of ".ngo" or ".cfg" extensions. The configuration data includes the name of installed file, an injection process, FTP and CnC information, mutex name. (Villeneuve and Bennett 2014). The communication between the infected machine and the attacker happens in a way similar to the one of the DarkComet, which is outlined in Figure 14.
The functionality of Xtreme allows the attacker to (Villeneuve and Bennett 2014):

- Read and modify the registry
- Interact via the remote shell
- Desktop capturing
- Capture data from connected devices, such as a microphone, webcam, etc.
- Manipulate running processes
- Upload and download files

4.1.9 CTB-Locker

The last malware family used was CTB-Locker, and it was represented by 79 unique files. This is another example of ransomware which encrypts user’s files asking for money for the decryption key. CTB is an acronym for Curve Tor Bitcoin, referring to Elliptic Curve algorithm that was used for encryption.

Figure 15. CTB-Locker operation
The propagation of the CTB-Locker samples was happening through the e-mails with malicious attachments. Attachments represented .zip files with the downloader inside. The initial operation of CTB-Locker is outlined in Figure 15. Upon execution malware drops itself to the %temp% folder with a random name and injects itself into the svchost.exe process. Moreover, a mutex of random name is created, ensuring that there is only one instance of CTB-Locker running on the machine.

Upon successful completion of malware, a pop-up screen will appear, providing information on payment and encryption details. This pop-up screen is shown in Figure 16. CTB-Locker targeted mostly Spain, France and Austria. (McAfee Labs 2015).

![CTB-Locker decryption instructions](image)

Figure 16. CTB-Locker decryption instructions (McAfee Labs 2015).

### 4.2 Cuckoo Sandbox

The study is based on and targeted to Cuckoo Sandbox. It is clear that to apply the machine learning algorithms to any problem, it is essential to represent the data in some form. For this purpose, Cuckoo Sandbox was used. The reports generated by the sandbox, describing the behavioral data of each sample, were preprocessed, and malware features were extracted from there. However, it is
important to understand the functionality of the sandbox and the structure of the reports first.

Cuckoo Sandbox is the open-source malware analysis tool that allows getting the detailed behavioral report of any file or URL in a matter of seconds. According to Cuckoo Foundation (2015), currently, supported file formats include:

- Generic Windows executables
- DLL files
- PDF documents
- Microsoft Office documents
- URLs and HTML files
- PHP scripts
- CPL files
- Visual Basic (VB) scripts
- ZIP files
- Java JAR
- Python files
- Almost anything else

Cuckoo has a highly customizable modular architecture, allowing it to be used both as a standalone application as well as integrated into the larger frameworks.

The main components of Cuckoo’s infrastructure are a host machine (the management software) and a number of guest machines (virtual or physical machines for analysis). Its operation scenario is quite straightforward: as soon as the new file is submitted to the server, a virtual environment is dynamically allocated for it, the file is executed, and all the actions performed in the system are recorded.
As shown in Figure 17, the sandbox generates the report which outlines all the behavior of the file in the system. The report is represented as a JSON file, and currently, it is capable of detecting the following features (Cuckoo Foundation 2015):

- Traces of calls performed by all processes spawned by the malware
- Files being created, deleted and downloaded by the malware during its execution
- Memory dumps of the malware processes
- Network traffic trace in the PCAP format
- Screenshots that were taken during the execution of the malware
- Full memory dumps of the machines

After getting the behavior of the file, Cuckoo Sandbox makes a decision on the level of maliciousness of the file using some pre-defined signatures. This functional part of the sandbox is only interesting to us as the way to compare the performance of the machine learning methods to the currently implemented signature-based methods.
4.2.1 Scoring system

The Cuckoo analysis score is an indication of how malicious an analyzed file is. The score is determined by measuring how many malicious actions are performed. Cuckoo uses a set of summarized malicious actions, called signatures, to identify the malicious behavior. Each of these signatures has its score, which indicates the severity of the performed action.

In total, there are three levels of severity and all levels have their score of severity: 1 for low, 2 for medium and 3 for high. An example of a low severity signature is the action of performing a query on a computer name. An example of a medium severity signature is the creation of an executable file. An example of a high severity signature is the removal of a shadow copy.

During analysis, all actions are stored to be processed afterwards. In the end, multiple modules, including the signatures module, are used to examine the stored actions. The signatures module examines all the collected data and finds patterns that match a signature. If the signature matches, a counter is incremented by the score of the severity of the signature (1, 2, or 3). When all signatures have been processed, the value of the counter is divided by 5.0 to create a floating point score. This score is the Cuckoo analysis score. An example of the signatures of different severity can be found in Figure 18.

Figure 18. Severity levels of cuckoo signatures
The average scores of the malware families used in this project are outlined in Table 2. The color indicates the maliciousness level corresponding to the score.

<table>
<thead>
<tr>
<th>Family</th>
<th>Average Cuckoo score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign</td>
<td>1.04</td>
</tr>
<tr>
<td>Dridex</td>
<td>5.26</td>
</tr>
<tr>
<td>Locky</td>
<td>6.41</td>
</tr>
<tr>
<td>Teslacrypt</td>
<td>6.27</td>
</tr>
<tr>
<td>Vawtrak</td>
<td>2.66</td>
</tr>
<tr>
<td>Zeus</td>
<td>6.46</td>
</tr>
<tr>
<td>DarkComet</td>
<td>5.15</td>
</tr>
<tr>
<td>CyberGate</td>
<td>6.57</td>
</tr>
<tr>
<td>Xtreme</td>
<td>5.15</td>
</tr>
<tr>
<td>CTB-Locker</td>
<td>4.76</td>
</tr>
</tbody>
</table>

Table 2. Cuckoo scores for malware families

It is hard to measure the accuracy of the detection since there is no threshold value indicating whether the sample is malicious or not. Moreover, determining the specific class to which malware belongs is beyond the functionality of the sandbox. In the graphical user interface, there are indicators of green, yellow and red colors, outlined in Figures 18 - 19, indicating how reliable the file is. The green indicator is used for samples with a score of 4 and lower, yellow for samples with score 4-7, red for scores 7-10. However, this feature is only an interface part and is not very reliable, as it is still in the alpha state. Moreover, it has some bugs, as outlined in Figure 20.
4.2.2 Reports and features

To apply machine learning algorithms to the problem, we need to figure out what kind of data should be extracted and how it should be presented.

Some works in the field are utilizing string properties or file formats properties as a basis for feature representation. For example, for Windows-based malware samples, the data contained in PE headers is often used as a base for analysis. However, implementing format-specific feature extraction is not the best solution, since formats of analyzed files can vary dramatically. (Hung 2011).

Other works rely on the so-called n-grams. Byte n-grams are overlapping substrings, collected in a sliding-window fashion where the windows of fixed size slides one byte at a time. Word n-grams and character n-grams are widely used in natural language processing, information retrieval, and text mining. (Reddy and Pujari 2006).

However, such approach has several disadvantages. The major difficulty in considering byte n-grams as a feature is that the set of all byte n-grams obtained from the set of byte strings of malware as well as of the benign programs is very large and it is not useful to apply classification techniques directly on these. (Reddy and Pujari 2006). In addition to that, such approach limits the ability of detection of polymorphic malware. In this case, the samples generating the same behavior will result in different strings, and, therefore, different n-grams.

Because of the above-mentioned reasons, in this study, it is decided to rely on the actual behavior of the files, that is monitored by the sandbox. Overall, we can identify the following features extracted by the sandbox:

- Files
- Registry keys
- Mutexes
- Processes
- IP addresses and DNS queries
- API calls

This section discusses which of the above-mentioned features should be used in our work.
• **Files**
  The reports contain information about opened files, written files, and created files. This kind of information is good in predicting the malware family since any malware files trigger many modifications to the file systems. It can be used for the quite accurate malware classification in most cases. However, for example in the cases of ransomware, relying solely on the file modifications might result in the algorithm not being able to distinguish different families. This is because ransomware encrypts every file on the system. Therefore the feature set consists mostly of the encrypted files. The differences between ransomware families would be defined by the files with malware settings, the amount of which is vastly lower than the whole feature set and, therefore, it would be very hard to make predictions based on this data.

• **Registry keys**
  On Windows systems, the registry stores the low-level system settings of the operation system and its applications. Any sample that is run on the system triggers a high amount of the registry changes – the Cuckoo reports can outline the registry keys opened, read, written, deleted. The information on the registry modifications can be a good source of information on the system changes caused by malware and can be used for malware detection.

• **Mutexes**
  The mutex stands for the Mutual Exclusion. This is a program object that allows multiple threads to share the same resource. Every time a program is started, a mutex with a unique name is created. Mutex names can be good identifiers of specific malware samples. However, for the families, they cannot result in the accurate result on a large scale, since the number of mutexes created per sample is dramatically lower than the dataset. That is why the small change related to the bug or non-started process would result in the dramatic change of the prediction results.

• **Processes**
  Common identifier of the specific malware sample is the name of the created process. However, very rarely it can be used for identification of
the malware family since in the common cases the process names are the same as the hash of the sample. As an alternative, the malware sample can inject itself into the system process. That is why this feature is bad for the family identification.

- **IP addresses and DNS queries**
  Cuckoo provides information about the network traffic in the PCAP format, from which the data about contacted IP addresses as well as DNS queries can be extracted. This data accurately identifies the IP addresses of the command and control servers of attackers and, therefore, can accurately identify the malware family in most cases. However, often the attackers change the domain names or IP addresses of their servers or spoof them. Therefore, it is unreliable to rely solely on this kind of information.

- **API calls**
  API stands for Application Programming Interface and refers to the set of tools that provide an interface for communication between different software components. API calls are recorded during the execution of the malware and refer to the specific process. They outline everything happening to the operating system, including the operations on the files, registry, mutexes, processes and other features mentioned earlier. For example, API calls OpenFile, OpenFileEx, CreateFile, CopyFileEx, etc. define the file operations, calls OpenMutex, CreateMutex and CreateMutexEx describe mutexes opened and created, etc. API call traces present the wide description of the sample behavior, including all the properties mentioned above. In addition to that, they include a wide set of distinct values. Moreover, they are simple to describe in numeric format, and that is why they were chosen as features. Here, the feature set will be defined by the number of unique API calls and the return codes. The next section describes the representation way in more detail.

### 4.3 Feature representation

Having familiarized ourselves with the features presented in the Cuckoo Sandbox reports, we can now think about the way to represent the features to be used for the machine learning algorithms. Since the feature set, containing
the failed and successful APIs as well as the return codes, is quite large, we have to find a way to present it in a clear, compact and non-redundant way. The representation chosen for this task is the Frequency (Binary) matrix, discussed in detail in the following section.

4.3.1 Binary representation

The binary representation is the most simple and straightforward way to represent the features of the failed and successful API calls. Here, a matrix is created, where the rows represent the samples, and the columns represent the API calls. A value of 0 represents the ‘failed’ state of the API call, and the value of 1 represents the successful API call.

\[
\text{API}_{\text{bin}} = \begin{bmatrix}
S_1 & API_1 & API_2 & \ldots & API_n \\
1 & 1 & \ldots & 0 \\
1 & 0 & \ldots & 1 \\
\vdots & \vdots & \ddots & \vdots \\
1 & 1 & \ldots & 1 \\
\end{bmatrix}
\]

Although this approach is simple and straightforward, it does not take into account the return codes generated, as well as a number of times the certain API call was triggered, resulting in lower accuracy. (Pirscoveanu 2015).

4.3.2 Frequency representation

The frequency representation approach is close to the binary representation approach in its structure. However, instead of marking each API call as ‘failed’ or ‘successful’, it outlines the frequency of each API call, showing a number of times it was triggered.

\[
\text{API}_{f_{\text{req}}} = \begin{bmatrix}
S_1 & API_1 & API_2 & \ldots & API_n \\
112 & 312 & \ldots & 72 \\
16 & 23 & \ldots & 315 \\
\vdots & \vdots & \ddots & \vdots \\
157 & 1 & \ldots & 567 \\
\end{bmatrix}
\]

Here, the horizontal axis represents the samples and the vertical axis represents the API call, where each number represents a number of times the
API call was triggered. This approach clearly provides more details than the binary representation, resulting in better accuracy. (Pirscoveanu 2015).

### 4.3.3 Combining representation

To utilize the maximum amount of useful data presented in the API calls information, the best approach is to combine the features of the previous representation methods. The resulting matrix would outline the frequency of failed APIs, successful APIs, and the return codes.

\[
\begin{bmatrix}
S_1 & Pass_1 & \ldots & Pass_n & Fail_1 & \ldots & Fail_n & RetC_1 & \ldots & RetC_n \\
23 & \ldots & 3 & 224 & \ldots & 123 & 23 & \ldots & 27 \\
52 & \ldots & 21 & 224 & \ldots & 57 & 224 & \ldots & 1 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
52 & \ldots & 22 & 210 & \ldots & 46 & 72 & \ldots & 111 \\
\end{bmatrix}
\]

Here the rows represent the samples, the columns \(Pass_1 \ldots Pass_n\) represent a number of times each API call in \([Pass_1; Pass_n]\) was called, where \(n\) is a total number of API calls triggered. Similarly, columns \(Fail_1 \ldots Fail_n\) represent a number of times each API call failed. Columns \(RetC_1 \ldots RetC_n\) represent a number of times each return code was returned. (Pirscoveanu 2015).

This approach results in a fair performance, and that is why it is chosen for our problem. Obviously, the usage of the combination method resulted in the dramatic increase in the number of features, since they are now represented by the combination of passed APIs, failed APIs and return codes, instead of relying solely on the APIs triggered. Since the feature set became more than two times bigger, some feature selection should be performed.

### 4.4 Feature selection

The goal of the feature selection is to remove the non-important features from the feature set as it gets too big. Bigger feature sets are harder to operate with, but some features in this set might not put any weight on the decision of the algorithm and, therefore, can be removed. For example, in our case, some API call might only be triggered in one sample once. In a case of a wide and variate feature set, this unique API call will not play any role in the algorithm and, therefore, removing it will not affect an accuracy in any way.
After extracting the features and representing them as a combination matrix, we ended up with 70518 features. This amount is too large for processing and accurate predictions. For example, with such a large feature set, it takes approximately two-three hours to load the dataset, preprocess it and run the k-nearest neighbors algorithm on an x64 8GB RAM machine. This amount of resources is unacceptable, and there is a need for removing irrelevant features.

Three general classes of feature selection methods are filtering methods, wrapper methods, and embedded methods (Guyon and Elisseef 2006).

- **Filter methods**
  Filter methods statistically score the features. The features with higher scores are kept in the dataset, while the features with the low scores are removed.

- **Wrapper methods**
  Here, the different feature combinations are tried with a prediction model and the combination that leads to the highest accuracy are chosen.

- **Embedded methods**
  These methods evaluate the features used while the model is being created.

### 4.5 Implementation

During this step the research plan is designed and can be implemented in practice.

The whole implementation process can be outlined in the following steps:

1. Sandbox configuration
2. Feature extraction (using Python 2.7)
3. Feature selection (using R)
4. Application of the machine learning methods (using R)
5. Evaluation of the results

Each of these steps is discussed in detail further in this chapter.
4.5.1 Sandbox configuration

To get the malware behavioral reports and to ensure that malware runs correctly, including all of its functionality, it is important to configure Cuckoo Sandbox. In the real world different malware samples exploit different vulnerabilities that might be part of certain software products. Therefore, it is important to include a broad range of services in the virtual machines created by the sandbox.

The hypervisor used for the virtual machines for Cuckoo is Virtualbox. The virtual machines will be created by using VMcloak, an automated virtual machine generation and cloaking tool for Cuckoo Sandbox. (Bremer 2015).

All virtual machines will have the following specifications:
- 1 CPU core 3.2 Ghz
- 2 GB RAM
- Internet connection

The installed software on all the virtual machines are:
- Windows 7 Professional 64bit without any updates, including Service Pack 1
- Adobe PDF reader 9.0
- Adobe Flashplayer 11.7.700.169
- Java JRE 7
- .NET framework 4.0

4.5.2 Feature extraction

As discussed in the previous section, the chosen feature representation method is the combining matrix that includes successful APIs, failed APIs and their return codes. This data is extracted from the reports generated by the sandbox.

The detailed process of feature extraction is outlined in Figure 21. In our implementation, the reports are stored locally after they were processed by the sandbox. Then, these reports are used as an input to the feature extraction script which produces the .csv file with the combining matrix inside. The number
of minimum API calls can be specified in the algorithm, e.g. all reports which triggered less than five API calls can be skipped. The file includes the timestamp of the extraction, and the logs, outlining the successful and unsuccessful operations are stored in a separate file.

Figure 21. Feature extraction process
4.5.3 Feature selection

As described in the previous chapter, feature selection is used for removing redundant and irrelevant features to improve the accuracy of the prediction. In our case, the feature set is extremely large, and the need for feature selection is, therefore, high.

The R language will be used for performing the feature selection and applying the machine learning methods. R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows, and MacOS. (Venables and Smith 2016).

A good and simple algorithm for feature selection in classification problems is the Boruta package. Roughly speaking, it is a wrapper method that works around the Random Forest algorithm. Its algorithm can be described as follows (Kursa and Rudnicki 2010):

1. Create shuffled copies of all features (to add more randomness). These are referred to as shadow copies.

2. Train a Random Forest classifier on the new dataset and apply a feature importance measure in the form of the Mean Decrease Accuracy algorithm. The importance of each feature is measured at this stage, and the weights are assigned.

3. On each iteration check if the feature from the initial feature set has a higher weight than the highest weight of this feature’s shadow copy. Remove the features that are ranked as unimportant at each iteration.

4. Stop after classifying all features as ‘selected’ or ‘rejected’, or after a certain number of iterations of random forest is achieved.

Unlike other feature selection methods, Boruta allows identifying all features that are somehow relevant to the result. Other methods, in turn, rely on a small feature subset that results in the minimal error. (Kursa and Rudnicki 2010).

The problem arises when we start implementing the feature selection. Having 70 518 features, the Boruta package exhausts, as it is not able to allocate enough memory and is not able to run. Therefore, we need to divide the dataset...
randomly into the subsets that can fit into the memory and run feature selection on all of them. Then, we collect all the features that were ranked as relevant and merge the subsets, leaving out all the non-important features. The next step is to run the feature selection again on the whole dataset. After running the feature selection algorithm, we ended up with 306 features. The performance of this change was evaluated based on the KNN accuracy with the given feature set. KNN was chosen for this problem, as it is the only algorithm that can process the whole feature set – it does not store any other information other than the dataset and does not build models, unlike other algorithms. After removing irrelevant features, the accuracy of detection based on KNN improved by approximately 1% and the prediction took approximately three seconds.

4.5.4 Application of machine learning methods

After the features were extracted and selected, we can apply the machine learning methods to the data that we obtained. The machine learning methods to be applied, as discussed previously, are K-Nearest Neighbours, Support Vector Machines, J48 Decision Tree, Naive Bayes, Random Forest. The general process is outlined in Figure 22.

The packages used for the implementation of algorithms are:

- K-Nearest Neighbours – class
- Support Vector Machines – kernlab
- J48 Decision Tree – RWeka
- Naive Bayes – e1071
- Random Forest – randomForest
- CrossTable plotting – gmodels
5 RESULTS AND DISCUSSION

This chapter discusses the results of the assessment of the implemented machine learning methods. The accuracy of detection is measured as the percentage of correctly identified instances:

\[
Accuracy = \frac{\text{count(Correctly identified samples)}}{\text{count(Total samples)}}
\]  

[15]

5.1 K-Nearest Neighbors

The result of the K-Nearest Method can be inferred from the cross table in Figure 23. The results outlined there should be understood as follows: rows represent the actual classes of the tested samples, while columns represent the predicted values. Therefore, the cell of the 1st row and 1st column will show the number of correct instances for the 1st class. The cell of the 1st row and 2nd column will show the number of 1st class instances, that were marked as 2nd class, etc.
As it can be seen, the test set consists of 371 samples, and 1 sample had an error resulting in a “0” class. The classification accuracy can be seen in Table 3.

<table>
<thead>
<tr>
<th>Class</th>
<th>Family</th>
<th>Correctly classified</th>
<th>Incorrectly classified</th>
<th>Accuracy</th>
<th>Average Cuckoo score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Benign</td>
<td>49</td>
<td>12</td>
<td>80.3%</td>
<td>1.04</td>
</tr>
<tr>
<td>2</td>
<td>Dridex</td>
<td>31</td>
<td>6</td>
<td>83.8%</td>
<td>5.26</td>
</tr>
<tr>
<td>3</td>
<td>Locky</td>
<td>22</td>
<td>5</td>
<td>81.5%</td>
<td>6.41</td>
</tr>
<tr>
<td>4</td>
<td>TeslaCrypt</td>
<td>43</td>
<td>1</td>
<td>97.7%</td>
<td>6.27</td>
</tr>
<tr>
<td>5</td>
<td>Vawtrak</td>
<td>15</td>
<td>3</td>
<td>83.3%</td>
<td>2.66</td>
</tr>
<tr>
<td>6</td>
<td>Zeus</td>
<td>30</td>
<td>10</td>
<td>75%</td>
<td>6.46</td>
</tr>
<tr>
<td>7</td>
<td>DarkComet</td>
<td>47</td>
<td>2</td>
<td>95.9%</td>
<td>5.15</td>
</tr>
<tr>
<td>8</td>
<td>CyberGate</td>
<td>38</td>
<td>0</td>
<td>100%</td>
<td>6.57</td>
</tr>
<tr>
<td>9</td>
<td>Xtreme</td>
<td>31</td>
<td>3</td>
<td>91.2%</td>
<td>5.15</td>
</tr>
<tr>
<td>10</td>
<td>CTB-Locker</td>
<td>17</td>
<td>5</td>
<td>77.3%</td>
<td>4.76</td>
</tr>
</tbody>
</table>

Table 3. KNN multi-class accuracy
The total accuracy of the K-Nearest Neighbors depends on the k value. In our case, different values were tested. They produced the following accuracy:

- k=1: 87%
- k=2: 84.63%
- k=3: 81.3%
- k=4: 80%
- k=5: 80%
- k=6: 80%
- k=10: 77.8%

As it can be seen, the best accuracy was achieved with k=1, and the accuracy was 87%. This is an unusual case – when the best accuracy is achieved with k=1 it can be a sign of one of the following:

1. The test data is the same as the training data.
2. The test data is very similar to the training data.
3. Boundaries between different classes are very clear.

In our case, the train and test set were selected randomly from the dataset with 2/3 ratio. This means that the data cannot be the same. The most probable reason for this is that the classes are distributed in a way that the boundaries are very clear when the KNN algorithm was applied.

Two-class classification into malware and benign files was also performed. The resulting cross-table can be seen in Figure 24. In the table, class 1 represents
the benign files, while class 2 represents malicious files. Again, predictions were made with different k values:

- k=1: 94.6%
- k=2: 94.3%
- k=3: 93.5%
- k=5: 93.5%
- k=7: 92.7%

The best accuracy was achieved with k=1 - 94.6%. The detailed accuracy can be found in the Tables 4.1 and 4.2.

<table>
<thead>
<tr>
<th>Class</th>
<th>Correctly identified instances</th>
<th>Incorrectly identified instances</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign</td>
<td>49</td>
<td>12</td>
<td>80.3%</td>
</tr>
<tr>
<td>Malicious</td>
<td>302</td>
<td>8</td>
<td>97.4%</td>
</tr>
</tbody>
</table>

Table 4.1. KNN binary accuracy

<table>
<thead>
<tr>
<th>True positives</th>
<th>True negatives</th>
<th>False positives</th>
<th>False negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>302</td>
<td>49</td>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4.2. KNN binary accuracy

Overall, the KNN algorithm resulted in a good accuracy of 87% for multi-class classification and 94.6% for two-class classification. We can conclude that the algorithm provided good results. Classes are distributed evenly in the case of multi-class classification, which also affected the good accuracy of the predictions. Even though the distribution is not even in the case of two-class classification (310 vs. 61), the results are still accurate.

### 5.2 Support Vector Machines

The next algorithm that was tested was Support Vector Machines. The result of the predictions can be outlined in Figure 25. The overall accuracy achieved was 87.6% for multi-class classification and 94.6% for binary classification.
The detailed information about the accuracy of each class can be found in Table 5.

<table>
<thead>
<tr>
<th>Class</th>
<th>Family</th>
<th>Correctly classified</th>
<th>Incorrectly classified</th>
<th>Accuracy</th>
<th>Average Cuckoo score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Benign</td>
<td>56</td>
<td>5</td>
<td>91.8%</td>
<td>1.04</td>
</tr>
<tr>
<td>2</td>
<td>Dridex</td>
<td>32</td>
<td>5</td>
<td>86.5%</td>
<td>5.26</td>
</tr>
<tr>
<td>3</td>
<td>Locky</td>
<td>21</td>
<td>6</td>
<td>77.8%</td>
<td>6.41</td>
</tr>
<tr>
<td>4</td>
<td>TeslaCrypt</td>
<td>37</td>
<td>7</td>
<td>84%</td>
<td>6.27</td>
</tr>
<tr>
<td>5</td>
<td>Vawtrak</td>
<td>10</td>
<td>8</td>
<td>55.6%</td>
<td>2.66</td>
</tr>
<tr>
<td>6</td>
<td>Zeus</td>
<td>31</td>
<td>9</td>
<td>77.5%</td>
<td>6.46</td>
</tr>
<tr>
<td>7</td>
<td>DarkComet</td>
<td>48</td>
<td>1</td>
<td>98%</td>
<td>5.15</td>
</tr>
<tr>
<td>8</td>
<td>CyberGate</td>
<td>37</td>
<td>1</td>
<td>97.4%</td>
<td>6.57</td>
</tr>
<tr>
<td>9</td>
<td>Xtreme</td>
<td>31</td>
<td>3</td>
<td>91.2%</td>
<td>5.15</td>
</tr>
<tr>
<td>10</td>
<td>CTB-Locker</td>
<td>22</td>
<td>0</td>
<td>100%</td>
<td>4.76</td>
</tr>
</tbody>
</table>

Table 5. SVM multiclass accuracy
Figure 26 outlines the cross-table for binary classification. The detailed information about binary classification can be found as well in Tables 6.1 and 6.2. As we can see, the number of correctly identified benign instances (true negatives) was equal to 41, correctly identified malicious instances (true positives) – 310, incorrectly identified benign instances (false positives) – 20, incorrectly identified malicious instances (false negatives) – 0.

<table>
<thead>
<tr>
<th>Class</th>
<th>Correctly identified instances</th>
<th>Incorrectly identified instances</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign</td>
<td>41</td>
<td>20</td>
<td>67.2%</td>
</tr>
<tr>
<td>Malicious</td>
<td>310</td>
<td>0</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 6.1. SVM binary classification accuracy

<table>
<thead>
<tr>
<th>True positives</th>
<th>True negatives</th>
<th>False positives</th>
<th>False negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>310</td>
<td>41</td>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2. SVM binary classification accuracy
Overall, the resulted accuracies of 87.6% for multi-class classification and 94.6% for binary classification are almost equal to the results of the K-Nearest Neighbors. In turn, this algorithm resulted in 0 false negatives in binary classification – this means that no malware samples were identified as benign. Therefore, it can prevent malware infections more effectively than K-Nearest Neighbors.

5.3 J48 Decision Tree

The third tested algorithm was the J48 Decision Tree. The advantage of the Decision Tree method is that it operates in the "white box" approach and we can see which decisions resulted in our prediction. The decision trees for multi-class classification and binary classification can be found in Figures 27 and 28 respectively.
Figure 27. Multiclass Decision Tree
Figure 28. Binary Decision Tree
The overall accuracy was 93.3% for multiclass classification and 94.6% for binary classification. The cross-table outlining the results of multiclass classification can be found in Figure 29.

<table>
<thead>
<tr>
<th>Predictions</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>Now total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4.40</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000000</td>
</tr>
<tr>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000000</td>
</tr>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000000</td>
</tr>
<tr>
<td>3</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000000</td>
</tr>
<tr>
<td>4</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000000</td>
</tr>
<tr>
<td>5</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000000</td>
</tr>
<tr>
<td>6</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000000</td>
</tr>
<tr>
<td>7</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000000</td>
</tr>
<tr>
<td>8</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000000</td>
</tr>
<tr>
<td>9</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000000</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.0000000</td>
</tr>
</tbody>
</table>

Figure 29. Decision Tree multi-class CrossTable

The detailed results of each malware family can be found in Table 7.
<table>
<thead>
<tr>
<th>Class</th>
<th>Family</th>
<th>Correctly classified</th>
<th>Incorrectly classified</th>
<th>Accuracy</th>
<th>Average Cuckoo score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Benign</td>
<td>54</td>
<td>7</td>
<td>88.5%</td>
<td>1.04</td>
</tr>
<tr>
<td>2</td>
<td>Dridex</td>
<td>37</td>
<td>0</td>
<td>100%</td>
<td>5.26</td>
</tr>
<tr>
<td>3</td>
<td>Locky</td>
<td>24</td>
<td>3</td>
<td>88.9%</td>
<td>6.41</td>
</tr>
<tr>
<td>4</td>
<td>TeslaCrypt</td>
<td>44</td>
<td>0</td>
<td>100%</td>
<td>6.27</td>
</tr>
<tr>
<td>5</td>
<td>Vawtrak</td>
<td>16</td>
<td>2</td>
<td>88.9%</td>
<td>2.66</td>
</tr>
<tr>
<td>6</td>
<td>Zeus</td>
<td>33</td>
<td>7</td>
<td>82.5%</td>
<td>6.46</td>
</tr>
<tr>
<td>7</td>
<td>DarkComet</td>
<td>47</td>
<td>2</td>
<td>95.9%</td>
<td>5.15</td>
</tr>
<tr>
<td>8</td>
<td>CyberGate</td>
<td>38</td>
<td>0</td>
<td>100%</td>
<td>6.57</td>
</tr>
<tr>
<td>9</td>
<td>Xtreme</td>
<td>32</td>
<td>2</td>
<td>94.1%</td>
<td>5.15</td>
</tr>
<tr>
<td>10</td>
<td>CTB-Locker</td>
<td>21</td>
<td>1</td>
<td>95.5%</td>
<td>4.76</td>
</tr>
</tbody>
</table>

Table 7. Decision Tree multi-class accuracy

For the binary classification problem, the algorithm resulted in 46 correctly identified instances for benign samples (true negatives), 305 correctly identified malware samples (true positives), 15 incorrectly identified benign samples (false positives) and 5 incorrectly classified benign samples (false negatives). The details are introduced in Figure 30 and Tables 8.1 and 8.2.

![CrossTable](image)

Figure 30. Decision Tree binary classification CrossTable

<table>
<thead>
<tr>
<th>Class</th>
<th>Correctly identified instances</th>
<th>Incorrectly identified instances</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign</td>
<td>46</td>
<td>15</td>
<td>75.4%</td>
</tr>
<tr>
<td>Malicious</td>
<td>305</td>
<td>5</td>
<td>98.4%</td>
</tr>
</tbody>
</table>

Table 8.1. Decision Tree binary classification accuracy
The overall accuracy of J48 Decision Tree was good: 93.3% for multiclass classification and 94.6% for binary classification. For multiclass classification, this result is sufficiently better than the one obtained with the K-Nearest Neighbors and Support Vector Machines. For binary classification, the result is the same, however.

5.4 Naive Bayes

The fourth algorithm that was tested was Naive Bayes. The resulted accuracy was 72.23% for multiclass classification and 55% for binary classification. The cross table related to the Naive Bayes classification can be found in Figure 31.
The detailed results that outline the accuracy of each of the malware families can be found in Table 9.

<table>
<thead>
<tr>
<th>Class</th>
<th>Family</th>
<th>Correctly classified</th>
<th>Incorrectly classified</th>
<th>Accuracy</th>
<th>Average Cuckoo score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Benign</td>
<td>34</td>
<td>27</td>
<td>55.8%</td>
<td>1.04</td>
</tr>
<tr>
<td>2</td>
<td>Dridex</td>
<td>1</td>
<td>36</td>
<td>2.7%</td>
<td>5.26</td>
</tr>
<tr>
<td>3</td>
<td>Locky</td>
<td>25</td>
<td>2</td>
<td>92.6%</td>
<td>6.41</td>
</tr>
<tr>
<td>4</td>
<td>TeslaCrypt</td>
<td>33</td>
<td>11</td>
<td>75%</td>
<td>6.27</td>
</tr>
<tr>
<td>5</td>
<td>Vawtrak</td>
<td>8</td>
<td>10</td>
<td>44.4%</td>
<td>2.66</td>
</tr>
<tr>
<td>6</td>
<td>Zeus</td>
<td>28</td>
<td>12</td>
<td>70%</td>
<td>6.46</td>
</tr>
<tr>
<td>7</td>
<td>DarkComet</td>
<td>49</td>
<td>0</td>
<td>100%</td>
<td>5.15</td>
</tr>
<tr>
<td>8</td>
<td>CyberGate</td>
<td>37</td>
<td>1</td>
<td>97.4%</td>
<td>6.57</td>
</tr>
<tr>
<td>9</td>
<td>Xtreme</td>
<td>31</td>
<td>3</td>
<td>91.2%</td>
<td>5.15</td>
</tr>
<tr>
<td>10</td>
<td>CTB-Locker</td>
<td>22</td>
<td>0</td>
<td>100%</td>
<td>4.76</td>
</tr>
</tbody>
</table>

Table 9. Naive Bayes multi-class classification accuracy

For binary classification, the algorithm performed poorly. The number of correctly identified benign instances (true negatives) was 61, correctly identified malware instances (true positives) 143, incorrectly identified benign instances (false positives) 0, incorrectly identified malware instances (false negatives) 167. The detailed results can be found in Figure 32 and in Tables 10.1 and 10.2.

Figure 32. Naive Bayes binary classification cross-table
Overall, the Naive Bayes algorithm performed poorly. The accuracy of multiclass classification was 72.23% and of binary classification only 55%. This result is insusceptible for real world detection. In addition to that, a number of false negatives, in other words, malware files that were incorrectly marked as benign filed, reached 167 – 45% of the total number of files. In a real environment, such result would cause a huge malware epidemics in a short amount of time.

Most likely, such a bad accuracy is the result of having a high dependability between features. As we know, the main drawback of the Naive Bayes algorithm is that each feature is treated independently, although in most cases this cannot be true. In our case, most likely certain APIs are dependent on each other, i.e. API$_n$ cannot be triggered without API$_m$. That is the most probable reason of a bad result of the Naive Bayes algorithm.

### 5.5 Random Forest

The last algorithm that was implemented was the Random Forest algorithm. The algorithm resulted in a good accuracy of predictions, 95.69% for multi-class classification and 96.8% for binary classification. The cross-table related to the multiclass predictions can be found in Figure 33.
The detailed information about the performance of each class can be found in Table 11.

<table>
<thead>
<tr>
<th>Class</th>
<th>Family</th>
<th>Correctly classified</th>
<th>Incorrectly classified</th>
<th>Accuracy</th>
<th>Average Cuckoo score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign</td>
<td>1</td>
<td>58</td>
<td>3</td>
<td>95%</td>
<td>1.04</td>
</tr>
<tr>
<td>Dridex</td>
<td>2</td>
<td>35</td>
<td>2</td>
<td>94.6%</td>
<td>5.26</td>
</tr>
<tr>
<td>Locky</td>
<td>3</td>
<td>25</td>
<td>2</td>
<td>92.6%</td>
<td>6.41</td>
</tr>
<tr>
<td>TeslaCrypt</td>
<td>4</td>
<td>44</td>
<td>0</td>
<td>100%</td>
<td>6.27</td>
</tr>
<tr>
<td>Vawtrak</td>
<td>5</td>
<td>15</td>
<td>3</td>
<td>83.3%</td>
<td>2.66</td>
</tr>
<tr>
<td>Zeus</td>
<td>6</td>
<td>35</td>
<td>5</td>
<td>87.5%</td>
<td>6.46</td>
</tr>
<tr>
<td>DarkComet</td>
<td>7</td>
<td>49</td>
<td>0</td>
<td>100%</td>
<td>5.15</td>
</tr>
<tr>
<td>CyberGate</td>
<td>8</td>
<td>38</td>
<td>0</td>
<td>100%</td>
<td>6.57</td>
</tr>
<tr>
<td>Xtreme</td>
<td>9</td>
<td>34</td>
<td>0</td>
<td>100%</td>
<td>5.15</td>
</tr>
<tr>
<td>CTB-Locker</td>
<td>10</td>
<td>22</td>
<td>0</td>
<td>100%</td>
<td>4.76</td>
</tr>
</tbody>
</table>

Table 11. Random Forest multiclass classification accuracy
In the binary classification problem, the result achieved reached 96.8%. More specifically, the number of correctly identified benign instances (true negatives) reached 52, correctly identified malware instances (true positives) 307, incorrectly identified benign instances (false positives) 9, and incorrectly identified malware instances (false negatives) 3. The detailed information can be found in Figure 34 and Tables 12.1 and 12.2.

![Figure 34. Random Forest binary classification cross-table](image)

<table>
<thead>
<tr>
<th>Class</th>
<th>Correctly identified instances</th>
<th>Incorrectly identified instances</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benign</td>
<td>52</td>
<td>9</td>
<td>85.2%</td>
</tr>
<tr>
<td>Malicious</td>
<td>307</td>
<td>3</td>
<td>99%</td>
</tr>
</tbody>
</table>

Table 12.1. Random Forest binary classification accuracy

<table>
<thead>
<tr>
<th>True positives</th>
<th>True negatives</th>
<th>False positives</th>
<th>False negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>307</td>
<td>52</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 12.2. Random Forest binary classification accuracy

The Random Forest algorithm resulted in the highest accuracy among the other algorithms. It achieved 95.69% and 96.8% accuracy for multiclass and binary classifications respectively. However, some false negatives are still present – their number is equal to three.
Overall, the goals defined for this study were achieved. The desired feature extraction and representation methods were selected and the selected machine learning algorithms were applied and evaluated.

The desired feature representation method was selected to be the combined matrix, outlining the frequency of successful and failed API calls along with the return codes for them. This was chosen, because it outlines the actual behavior of the file. Unlike other methods, it combines information about different changes in the system, including the changes in the registry, mutexes, files, etc.

In classification problems, different models gave different results. The lowest accuracy was achieved by Naive Bayes (72.34% and 55%), followed by k-Nearest-Neighbors and Support Vector Machines (87%, 94.6% and 87.6%, 94.6% respectively). The highest accuracy was achieved with the J48 and Random Forest models, and it was equal to 93.3% and 95.69% for multi-class classification and 94.6% and 96.8% for binary classification respectively.

The result achieved by Random Forest is more accurate than the one achieved by the sandbox. It is hard to compare the results quantitively, since the sandbox does not classify the samples into malicious or benign. The classification into malware family is beyond its functionality as well. Instead, the maliciousness of the file is seen as a regression problem, and the severity score is its output. However, the difference in the accuracy can be easily seen. Table 2, outlined in Chapter 4.2.1, shows that none of the malware families were labeled with the “red” severity level, and one was labeled as “green”. This result is very inaccurate in comparison to the 95.69% and 96.8% achieved by Random Forest.

Based on the results described before, it is recommended to implement the classification based on the Random Forest method for multi-class classification, as it resulted in the best accuracy and high performance. Although this method achieved the highest result for binary classification as well, it is recommended to consider implementing Support Vector Machines instead. This is because this method resulted in 0 false-negatives, i.e. no malware samples were classified as benign. Although in the binary problem accuracy is still the main
concern, the number of false-negatives is an important factor as well, since they can result in massive infections. Random Forest, despite its high accuracy, resulted in 3 false negatives. Support Vector Machines, in turn, resulted in 0 false-negatives, while the accuracy is lower by only 2%. That is why it is recommended to consider implementing Support Vector Machines for binary classification.

<table>
<thead>
<tr>
<th></th>
<th>Classifier</th>
<th>KNN</th>
<th>SVM</th>
<th>Naive Bayes</th>
<th>J48</th>
<th>Random Forest</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Multi-class</strong></td>
<td>Accuracy</td>
<td>87%</td>
<td>87.6%</td>
<td>72.34%</td>
<td>93.3%</td>
<td>95.69%</td>
</tr>
<tr>
<td></td>
<td>False-positives</td>
<td>12</td>
<td>20</td>
<td>0</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>False-negatives</td>
<td>8</td>
<td>0</td>
<td>167</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>True-positives</td>
<td>302</td>
<td>310</td>
<td>143</td>
<td>305</td>
<td>307</td>
</tr>
<tr>
<td></td>
<td>True-negatives</td>
<td>49</td>
<td>41</td>
<td>61</td>
<td>46</td>
<td>52</td>
</tr>
</tbody>
</table>

Table 53. Results

6.1. **Future Work**

The study performed in this project was a proof-of-concept. Therefore, several future improvements related to the practical implementation of this project can be identified:

- **Implement feature extraction in the inline mode**
  Currently, the feature extraction is performed after the files were run in the sandbox and the reports were generated. This approach will result in delays in the file analysis when implemented. Instead, it is advised to
extract the features as they are processed by the sandbox, so that there will be no need to go through the reports again.

- **Use a wider dataset**
  Although the dataset that was used in this study is broad, covering most of the malware types that are relevant to the modern world, it does not cover all possible types. Collecting a malware dataset is a tedious task that requires a lot of time and effort. For more accurate evaluation of the predictors, it is advised to test the models on all the possible types of malware: spyware, adware, rootkits, backdoor, banking malware, etc. In addition to that, it is important to understand that the model will only be able to predict the samples of the families that it has seen earlier. In other words, in a real-world application, the maximum amount of possible families should be used before the launch of the project for real-world environments.

- **Use pre-selected APIs**
  In this work, the big overhead in the data processing was created by the need of selecting the relevant API calls and removing the redundant ones. For further implementation, only the APIs that were identified as relevant in this study can be used. This will decrease the amount of time required for data preprocessing, reduce the performance requirements of the machine on which the analysis is being done and decrease the level of feature selection to be made. However, it should be noted, that for more accurate description, the relevant APIs should be extracted from the biggest possible dataset. Also, it is advised to select the relevant APIs per malware family, as this will result in another level of flexibility and accuracy.
BIBLIOGRAPHY


Aliyev, Vusal. 2010. Using honeypots to study skill level of attackers based on the exploited vulnerabilities in the network. Chalmers University of Technology.


McAfee Labs. 2015. Threat Advisory CTB-Locker.


McAfee Labs. 2016. Threat Advisory TeslaCrypt Ransomware.


Pirsicoveanu, Radu-Stefan. 2015. Clustering Analysis of Malware Behavior. *Aalborg University*.


Smith, Craig, Ashraf Matrawy, Stanley Chow, and Bassem Abdelaziz. 2009.


APPENDICES

1. Feature Extraction Code (Python)

```python
#!/usr/bin/env python

import json
import logging
import os
import sys
import time
import datetime
import numpy

def get_json(filepath, log):
    """
    Reads a JSON file, returns None on ValueError
    and if not a file.
    """
    parsed_json = None
    # See if it is a file and not a directory or something else
    if not os.path.isfile(filepath):
        log.warning("%s is not a file! Skipping", filepath)
        return parsed_json
    try:
        with open(filepath, "rb") as fp:
            parsed_json = json.loads(fp.read())
    except ValueError as e:
        log.error("Error reading JSON file %s. Error: %s", filepath, e)
    return parsed_json

def setup_logger():
    """
    Sets up the logger.
    """
    logformat = "[%(asctime)s %(levelname)s] %(message)s"
    dateformat = "%d-%m-%y %H:%M:%S"
    logger = logging.getLogger("extraction")
    formatter = logging.Formatter(logformat)
    formatter.datefmt = dateformat
    fh = logging.FileHandler("dataextraction.log", mode="a")
    fh.setFormatter(formatter)
    sh = logging.StreamHandler()
    sh.setFormatter(formatter)
    logger.setLevel(logging.INFO)
    logger.addHandler(fh)
    logger.addHandler(sh)
    logger.propagate = False
```
if __name__ == "__main__":

    setup_logger()
    log = logging.getLogger("extraction")

    # Add path to reports here
    DATASET_DIR = "/home/kate/thesis/reports"

    # Location where data will be stored. This should be a directory, not a filename.
    # The filename will be generated using a timestamp. Format: data-
    NUMPY_DATA_SAVE = "/home/kate/thesis"

    if len(sys.argv) > 1:
        min_calls = int(sys.argv[1])
        log.info("-----| Using only reports with a minimum of %s
        calls | <------", min_calls)
    else:
        min_calls = 1
        log.info("-----| Using only reports with a minimum of 1
        call | <------")

    ignore_list = []

    success_apis, fail_apis, return_codes = [], [], []
    sample_num = 0

    # Fill the lists with calls
    for sample in os.listdir(DATASET_DIR):

        file_path = os.path.join(DATASET_DIR, sample)

        try:
            # Load JSON file
            log.info("Reading file: %s", file_path)
parsed_json = get_json(file_path, log)

            if parsed_json is None:
                log.warning("Parsed JSON was None. Skipping %s", file_path)
                continue

            # Successfully loaded, increment number
            sample_num += 1

            # Check the number of API calls
            total = 0
            for proc in parsed_json["behavior"]['processes']:
                total += len(proc['calls'])
            if total < min_calls:
                log.warning("Sample %s is less than %s calls. Skipping...", file_path, min_calls)
                ignore_list.append(file_path)
                continue

        except Exception as e:
            log.error("Error loading file: %s", file_path)
            continue

    # Successfully loaded, increment number
    sample_num += 1
for n in parsed_json['behavior']['processes']:
    for k in n['calls']:
        call = k['api']
        if k['status'] == 1:
            if call not in success_apis:
                success_apis.append(call)
        elif call not in fail_apis:
            fail_apis.append(call)
        if k['return_value'] not in return_codes:
            return_codes.append(k['return_value'])

except MemoryError as e:
    log.error("Error! %s", e)
    sys.exit(1)

log.info("Success APIs: %s", len(success_apis))
log.info("Fail APIs: %s", len(fail_apis))
log.info("Return codes: %s", len(return_codes))

data_length = len(success_apis) + len(fail_apis) + len(return_codes) + 2

# Create the matrix using the calculated length of all lists
matrix = numpy.zeros((sample_num, data_length))
matrix_scores = numpy.zeros((sample_num, 3))

log.info("Data length: %s", data_length)
ids = 0

try:
    for sample in os.listdir(DATASET_DIR):
        file_path = os.path.join(DATASET_DIR, sample)
        if file_path in ignore_list:
            continue
        # Load JSON file again
        log.info("Reading file %s", file_path)
        parsed_json = get_json(file_path, log)
        if parsed_json is None:
            log.warning("Parsed JSON was None. Skipping %s", file_path)
            continue
        log.info("Handling all calls for file %s", file_path)
        for process in parsed_json['behavior']['processes']:
            for call in process['calls']:
                status = call['status']
                api = call['api']
                returncode = call['return_value']
if status:
    # Store successful call in matrix
    index = success_apis.index(api)
    matrix[ids][index] += 1
else:
    # Store failed call in matrix
    offset = len(success_apis)
    index = offset + fail_apis.index(api)
    matrix[ids][index] += 1
    # Store return code in matrix
    offset = len(success_apis) + len(fail_apis)
    index = offset + return_codes.index(returncode)
    matrix[ids][index] += 1

if sample.split("-")[0] == "benign":
    matrix[ids][data_length-2] = 1
    matrix[ids][data_length-1] = 1
    matrix_scores[ids][2] = 1
elif sample.split("-")[0] == "dridex":
    matrix[ids][data_length-2] = 2
    matrix[ids][data_length-1] = 2
    matrix_scores[ids][2] = 2
elif sample.split("-")[0] == "locky":
    matrix[ids][data_length-2] = 3
    matrix[ids][data_length-1] = 2
    matrix_scores[ids][2] = 3
elif sample.split("-")[0] == "teslacrypt":
    matrix[ids][data_length-2] = 4
    matrix[ids][data_length-1] = 2
    matrix_scores[ids][2] = 4
elif sample.split("-")[0] == "vawtrak":
    matrix[ids][data_length-2] = 5
    matrix[ids][data_length-1] = 2
    matrix_scores[ids][2] = 5
elif sample.split("-")[0] == "zeus":
    matrix[ids][data_length-2] = 6
    matrix[ids][data_length-1] = 2
    matrix_scores[ids][2] = 6
elif sample.split("-")[0] == "darkcomet":
    matrix[ids][data_length-2] = 7
    matrix[ids][data_length-1] = 2
    matrix_scores[ids][2] = 7
elif sample.split("-")[0] == "cybergate":
    matrix[ids][data_length-2] = 8
    matrix[ids][data_length-1] = 2
    matrix_scores[ids][2] = 8
elif sample.split("-")[0] == "xtreme":
    matrix[ids][data_length-2] = 9
    matrix[ids][data_length-1] = 2
    matrix_scores[ids][2] = 9
elif sample.split("-")[0] == "ctblocker":
    matrix[ids][data_length-2] = 10
    matrix[ids][data_length-1] = 2
    matrix_scores[ids][2] = 10
matrix_scores[ids][0] = sample.split(".")[0].split("-")[1]
matrix_scores[ids][1] = parsed_json["info"]['score']

ids += 1

finally:
    date_time = time.time()
dt_stamp = datetime.datetime.fromtimestamp(time.time()).strftime('%d-%m-%Y_%H-%M-%S')
filename = "data-%s.csv" % dt_stamp
file_scores = "scores-%s.csv" % dt_stamp
try:
    path = os.path.join(NUMPY_DATA_SAVE, filename)
    path_scores = os.path.join(NUMPY_DATA_SAVE, file_scores)
    log.info("Storing numpy data in at %s", path)
numpy.savetxt(path, matrix, delimiter="",
numpy.savetxt(path_scores, matrix_scores, delimiter="","
except Exception as e:
    path = os.path.join(os.path.dirname(os.path.realpath(__file__)), filename)
    path_scores = os.path.join(os.path.dirname(os.path.realpath(__file__)),
    file_scores)
    log.error("Error writing numpy data! Trying script directory %s",
    path)
    numpy.savetxt(path, matrix, delimiter="",
    numpy.savetxt(path_scores, matrix_scores, delimiter="","

Note: after this script, the headers of “Class” and “Malware” should be added to the respective columns. The classes are derived from the names of files.
2. Feature selection code (R)

```r
library(RWeka)
library(Boruta)

home_dir <- "C:\Users\Kateryna\Desktop\Thesis-stuff\data-07-12-2016_18-13-37.csv"

# load data from .csv file
apis1 <- read.csv(home_dir, header=TRUE)

set.seed(123)
boruta.train <- Boruta(Class ~., data = apis1, doTrace = 2)
print(boruta.train)
final.boruta <- TentativeRoughFix(boruta.train)
print(final.boruta)
k1 = getSelectedAttributes(final.boruta, withTentative = F)
boruta.df <- attStats(final.boruta)

selected_apis1 <- apis1[, k1]
selected_apis1 <- cbind(selected_apis1, apis1[70518])
write.csv(selected_apis1, file = "selectedfeatures.csv")
```
3. Classification code (R)

```r
# import libraries
library(RWeka)
library(kernlab)
library(Boruta)
library(class)
library(dplyr)
library(lubridate)
library(gmodels)
library(ggvis)
library(e1071)
library(randomForest)

# set directory
home_dir <- "C:\\Users\\Kateryna\\Desktop\\Thesis-stuff\\selectedfeatures.csv"

# load data from .csv file
selected_apis <- read.csv(home_dir, header=TRUE)

# define the normalization function
normalize <- function(x) {
  num <- x - min(x)
  denom <- max(x) - min(x)
  return (num/denom)
}

coln = ncol(selected_apis)
coln1=coln+1

# normalize data
selected_apis<- as.data.frame(lapply(selected_apis[,1:coln-1], normalize))

# divide data into training and test set
set.seed(1234)
# label matrix with 1 with prob 0.067 and 2 with prob 0.33, to separate dataset into 2/3 ratio
ind <- sample(2, nrow(selected_apis), replace=TRUE, prob=c(0.67, 0.33))

# create 2 datasets from tables, without the class label
selected_apis.train1<-selected_apis[ind==1,(1:coln)]
selected_apis.test1<-selected_apis[ind==2,(1:coln)]
selected_apis.train2<-selected_apis[ind==1,1:coln1]
selected_apis.test2<-selected_apis[ind==2,1:coln1]
selected_apis.train2<-selected_apis.train2[-coln]
selected_apis.test2<-selected_apis.test2[-coln]

# set class labels for training and test sets
selected_apis.trainlabels<-selected_apis[ind==2,coln]
selected_apis.trainlabels.twoway<-selected_apis[ind==2,coln1]

# set class labels for 2-class classification
selected_apis.testlabels.twoway<-selected_apis[ind==2,coln1]
selected_apis.trainlabels.twoway<-selected_apis[ind==1,coln1]

# J48
```
fit <- J48(as.factor(Class)~., data=selected_apis.train1)
# summarize the fit
summary(fit)
# make predictions
predictions <- predict(fit, selected_apis.test1)
# summarize accuracy
CrossTable(selected_apis.testlabels, predictions, type="C-Classification")

#two-class
fit2 <- ksvm(as.factor(Malware)~., data=selected_apis.train2)
# summarize the fit
summary(fit2)
# make predictions
predictions <- predict(fit2, selected_apis.test2)
# summarize accuracy
CrossTable(selected_apis.testlabels.twoway, predictions, type="C-Classification")

#KNN classification
model_pred <- knn(train = selected_apis.train1, test = selected_apis.test1, cl = selected_apis.trainlabels, k=1)
CrossTable(x = selected_apis.testlabels, y = model_pred, prop.chisq=FALSE)

#two-way
model_twoway<-knn(train = selected_apis.train2, test = selected_apis.test2, cl = selected_apis.trainlabels.twoway, k=1)
prob <- attr(model_pred, "prob")
CrossTable(x = selected_apis.testlabels.twoway, y = model_twoway, prop.chisq=FALSE)

#Naive Bayes
fit <- naiveBayes(as.factor(Class)~., data=selected_apis.train1)
# summarize the fit
summary(fit)
# make predictions
predictions <- predict(fit, selected_apis.test1)
# summarize accuracy
CrossTable(selected_apis.testlabels, predictions, type="C-Classification")

#two-way
fit <- naiveBayes(as.factor(Malware)~., data=selected_apis.train2)
# summarize the fit
summary(fit)
# make predictions
predictions <- predict(fit, selected_apis.test2)
# summarize accuracy
CrossTable(selected_apis.testlabels, predictions, type="C-Classification")

#RandomForest
fit <- randomForest(as.factor(Class)~., data=selected_apis.train1)
# summarize the fit
summary(fit)
# make predictions
predictions <- predict(fit, selected_apis.test1)
# summarize accuracy
CrossTable(selected_apis.testlabels, predictions, type="C-Classification")

# two-way
fit <- randomForest(as.factor(Malware)~., data=selected_apis.train2)
# summarize the fit
summary(fit)
# make predictions
predictions <- predict(fit, selected_apis.test2)
# summarize accuracy
CrossTable(selected_apis.testlabels, predictions, type="C-Classification")

# SVM
# fit model
fit <- ksvm(as.factor(Class)~., data=selected_apis.train1)
# summarize the fit
summary(fit)
# make predictions
predictions <- predict(fit, selected_apis.test1)
# summarize accuracy
CrossTable(selected_apis.testlabels, predictions, type="C-Classification")

# two-way
fit <- ksvm(as.factor(Malware)~., data=selected_apis.train2)
# summarize the fit
summary(fit)
# make predictions
predictions <- predict(fit, selected_apis.test2)
# summarize accuracy
CrossTable(selected_apis.testlabels, predictions, type="C-Classification")
Zeus

CyberGate