Kateryna Chumachenko

MACHINE LEARNING METHODS FOR MALWARE DETECTION AND CLASSIFICATION

Bachelor’s Thesis
Information Technology

2017

University of Applied Sciences

Author (authors)	Degree	Time
Kateryna Chumachenko	Bachelor of Engineering	March 2017

Thesis Title	
Machine Learning Methods for Malware Detection and Classification	93 pages 14 pages of appendices

Commissioned by

Cuckoo Sandbox

Supervisor

Matti Juutilainen

Abstract

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 consistsed 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 wheher 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 permorms 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.

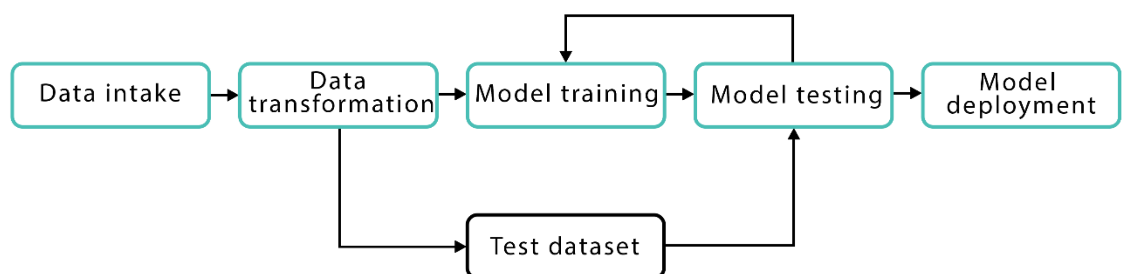


Figure 1. General workflow process

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*, 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 μ_i and σ_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.

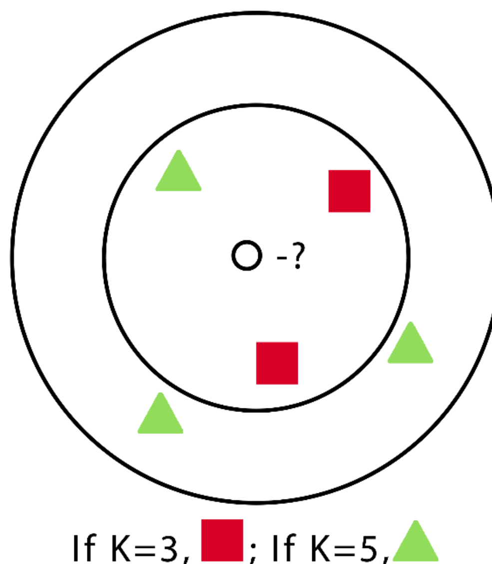


Figure 2. KNN example

Different distance measurement methods are used for finding the closest neighbors. The popular ones include Hamming Distance, Manhattan Distance, Minkowski distance:

$$Hamming\ Distance\colon\ d_{ij} = \sum_{k=1}^{p} |x_{ik} - x_{jk}| \quad [1]$$

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

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

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

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

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:

$$k = \sqrt{n} \quad [5]$$

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.

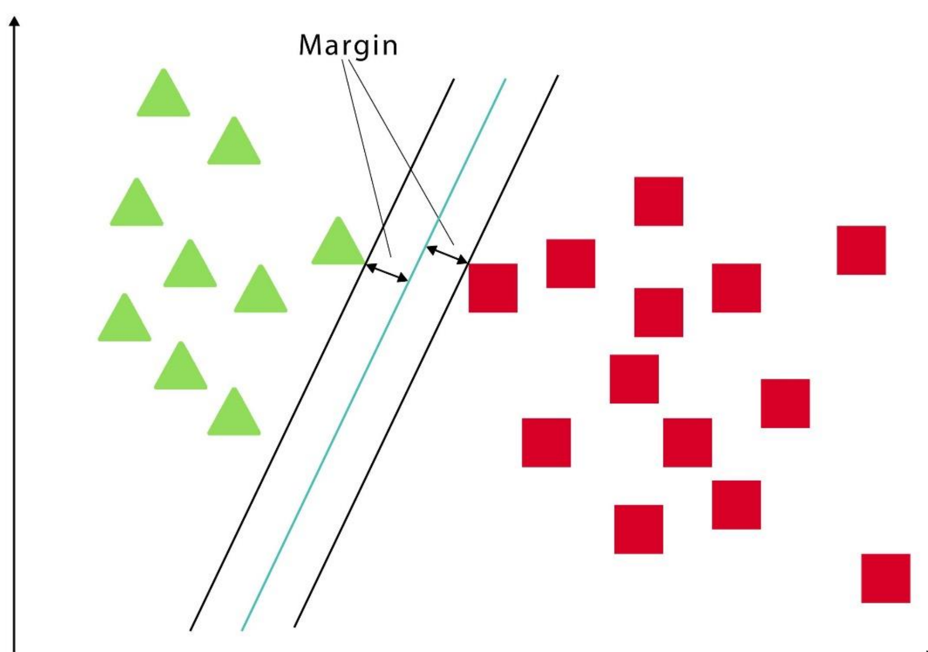


Figure 3. SVM scheme

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),…,(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.

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

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

$$R(a) = \int l(f(x,a),y)dP(x,y) = Test\ Error \qquad [7]$$

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

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

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:

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

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

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

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

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \quad [11]$$

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:

Predictors				Target
Outlook	Temperature	Humidity	Windy	Play tennis
Rainy	Hot	High	False	No
Rainy	Hot	High	True	No
Overcast	Hot	High	False	Yes
Sunny	Mild	High	False	Yes
Sunny	Cool	Normal	False	Yes
Overcast	Cool	High	True	No
Rainy	Cool	High	True	Yes
Rainy	Mild	High	False	No
Sunny	Cool	Normal	False	Yes

Table 1. Decision tree example dataset

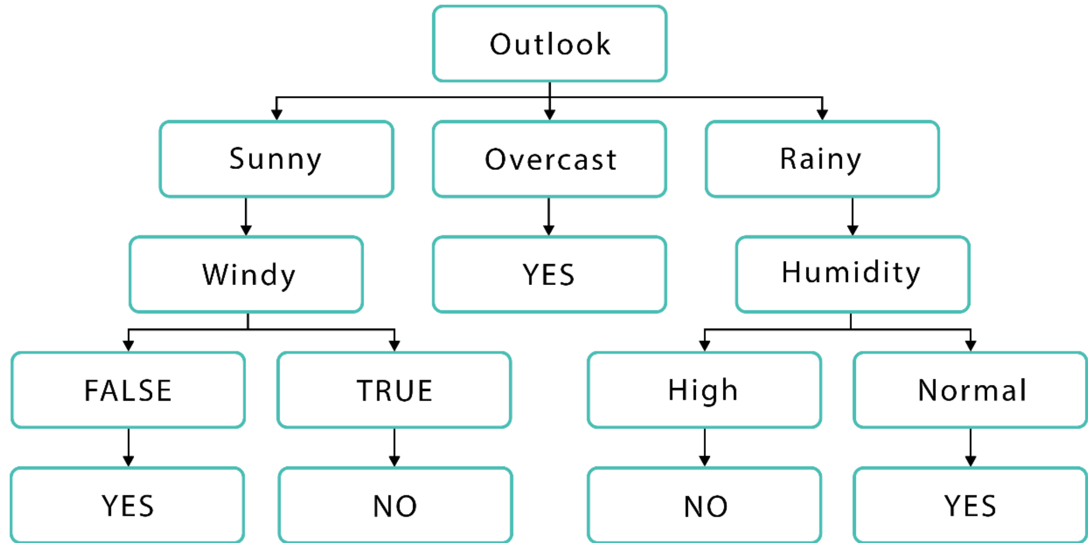


Figure 4. Decision tree example

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.

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

$$E(S) = \sum_{i=1}^{c} -p_i \log_2 p_i \quad [13]$$

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.

$$Gain(T,X) = Entropy(T) - Entropy(T,X) \qquad [14]$$

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.

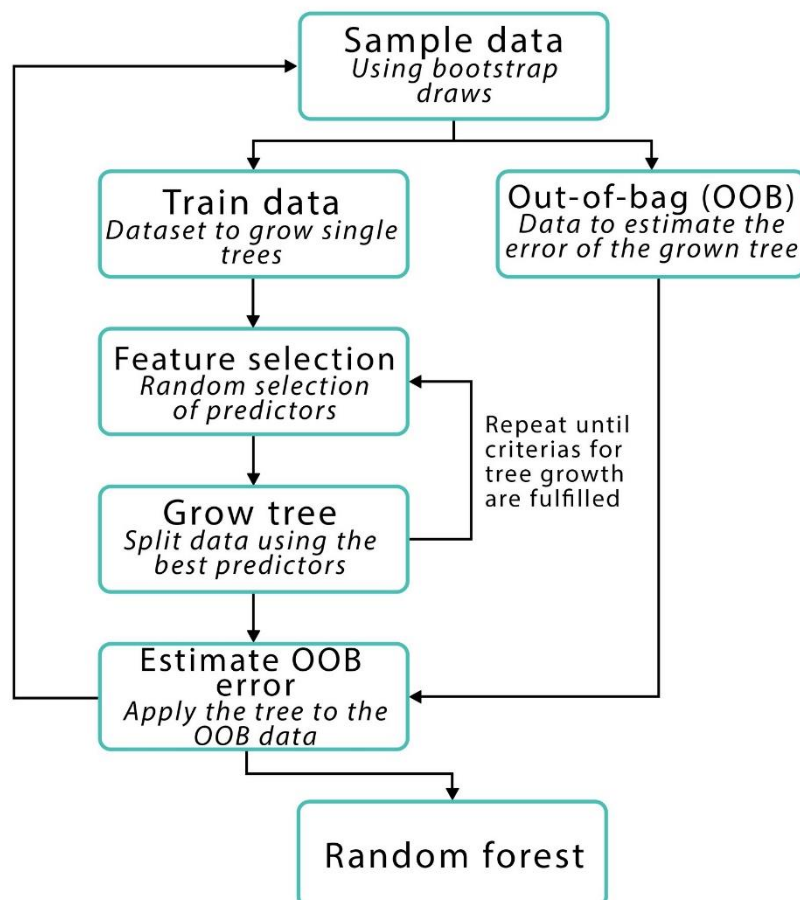


Figure 5. Random Forest scheme

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 metho**d 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.

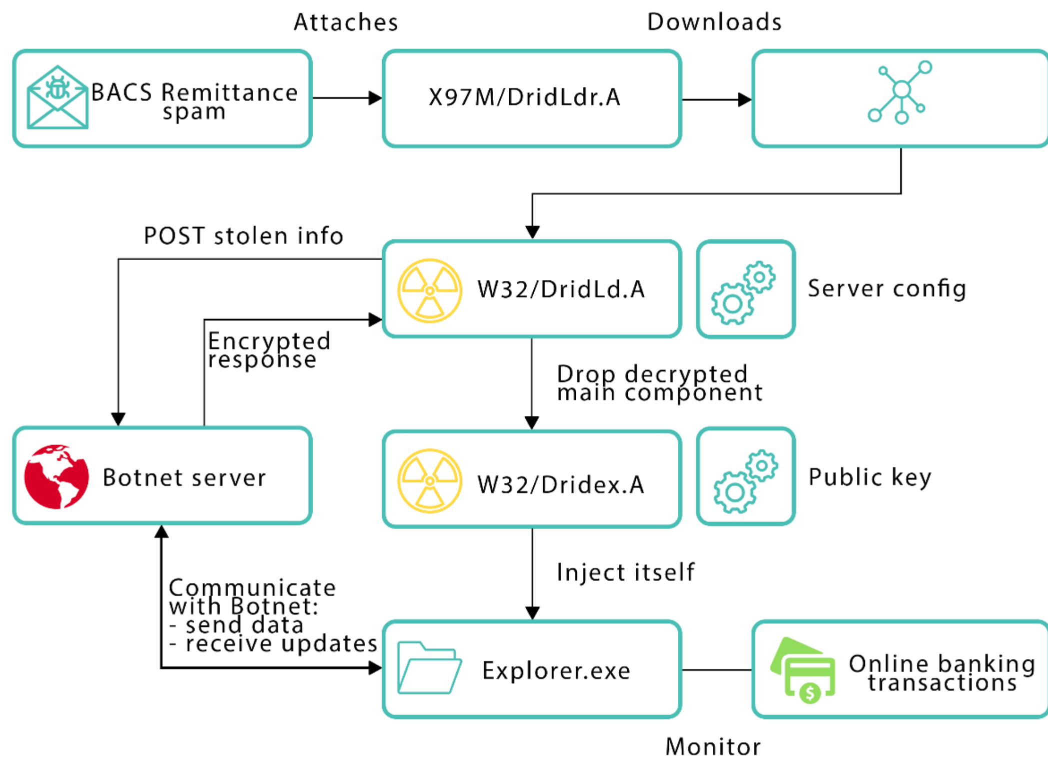


Figure 6. Dridex operation scheme (Aquino 2014)

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.

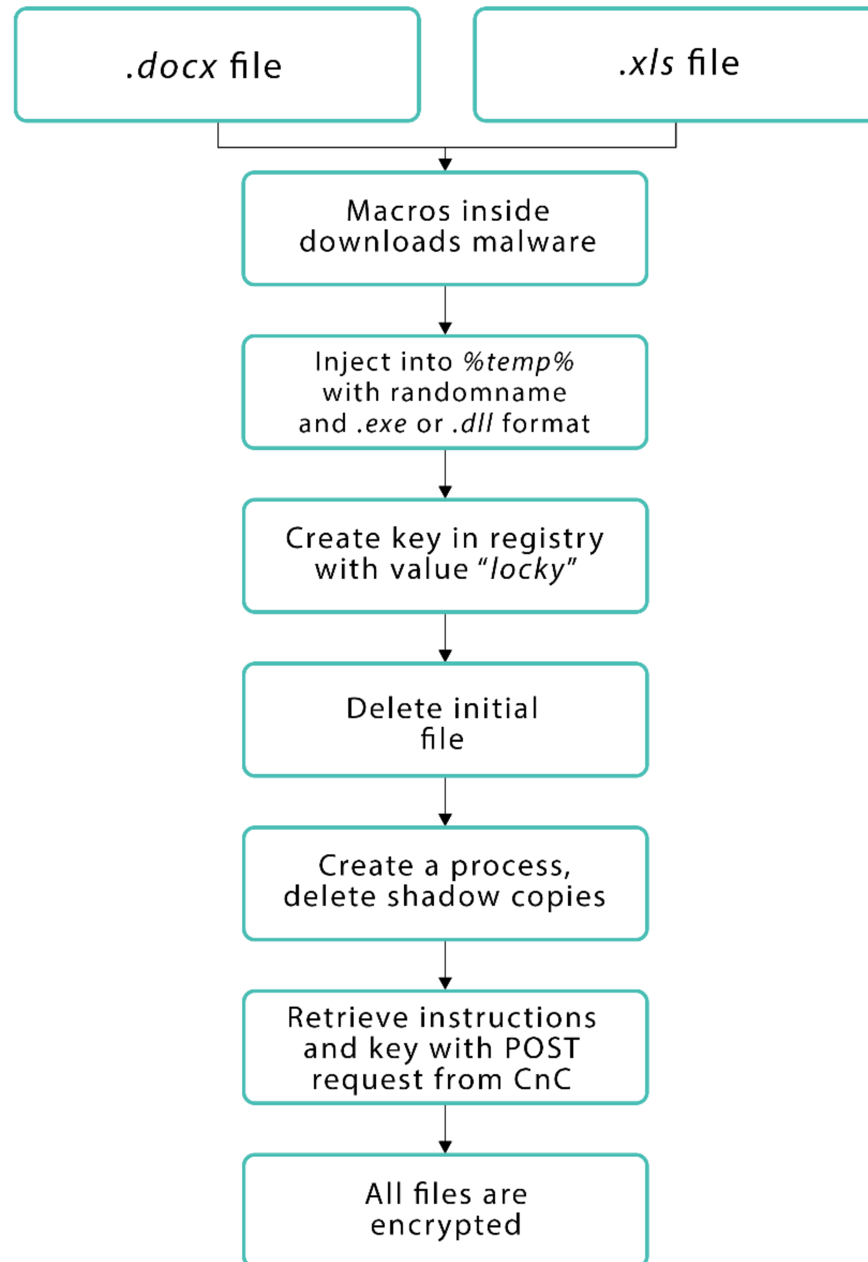


Figure 7. Locky operation

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.

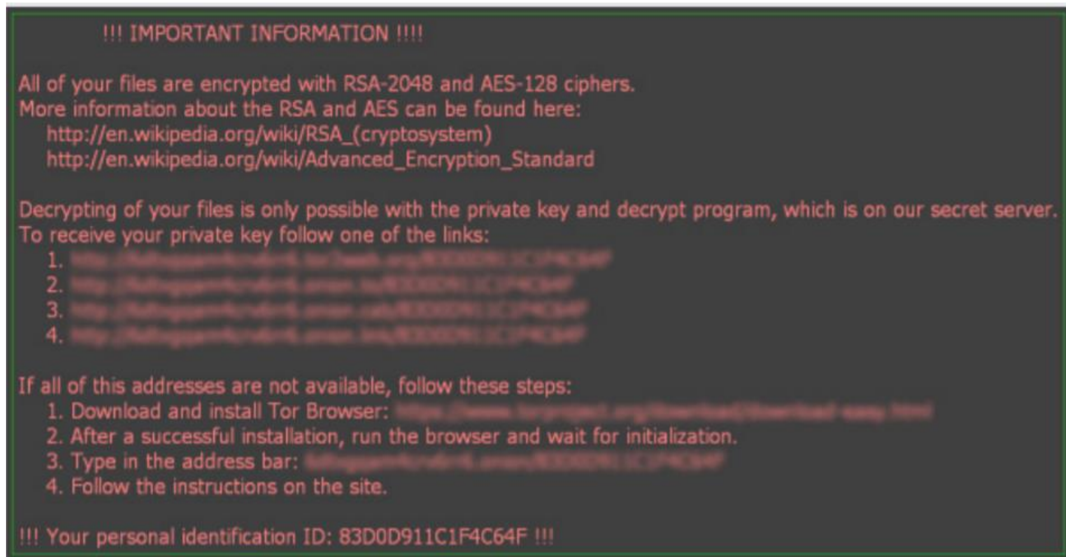


Figure 8. Recovery instructions of the Locky malware (Symantec Security Response 2016)

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.

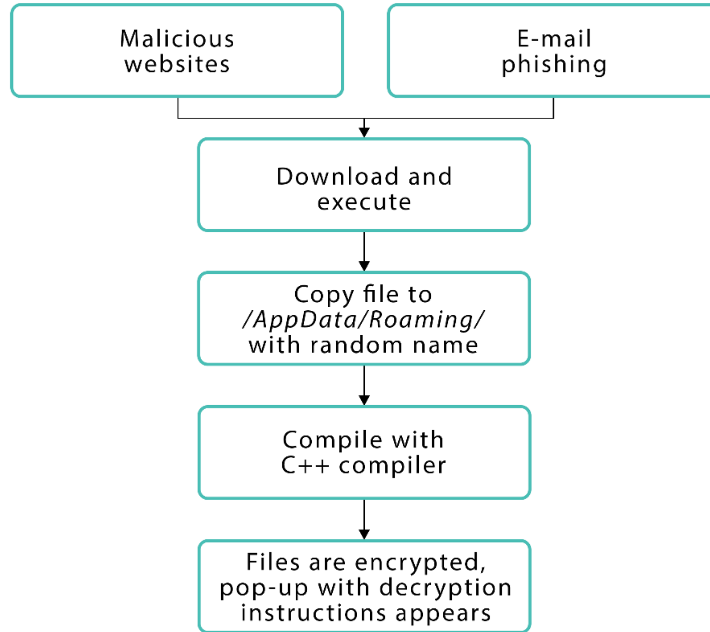


Figure 9. TeslaCrypt operation

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):

Figure 10. Decryption instructions for TeslaCrypt (McAfee Labs 2016)

Payment for a decryption key is requested to be made via PayPal or Bitcoin (1 000 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.

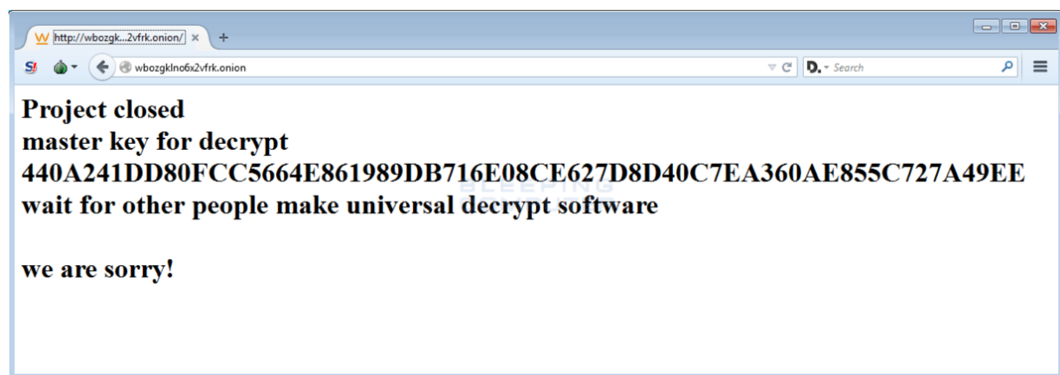


Figure 11. Payment page of TeslaCrypt with the master decryption key (Mimoso 2016)

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.

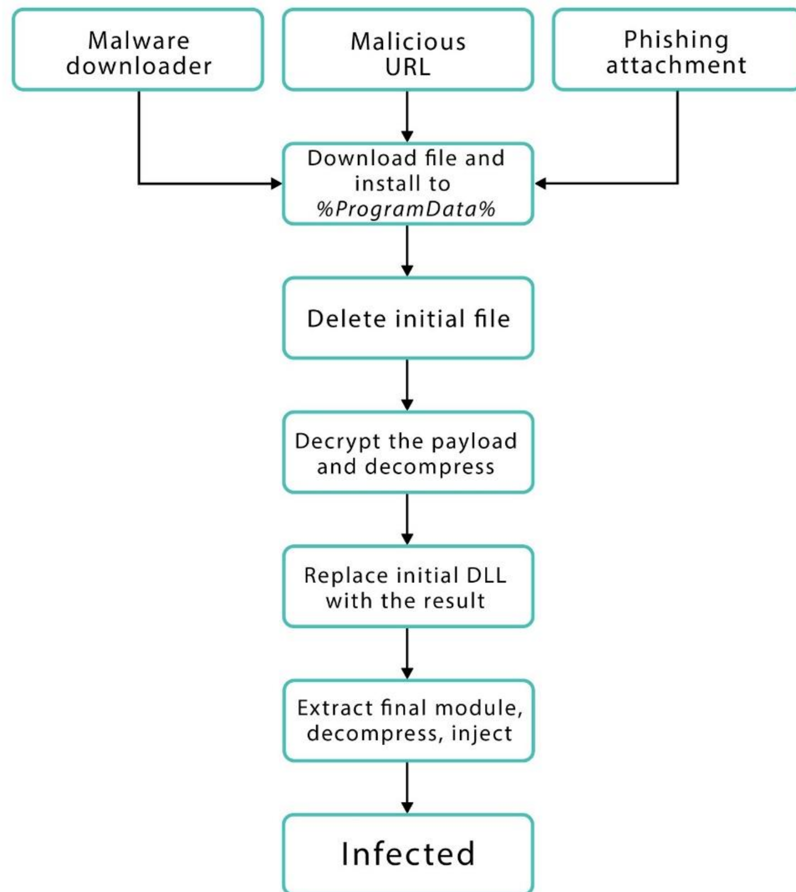


Figure 12. Vawtrak operation

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.

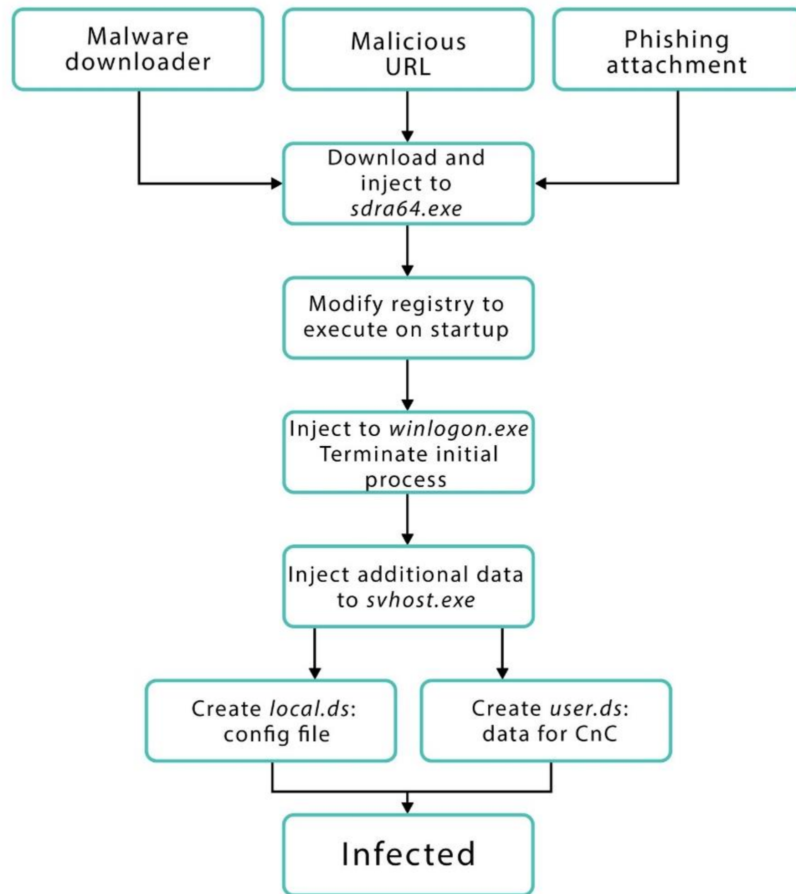


Figure 13. Zeus operation

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.

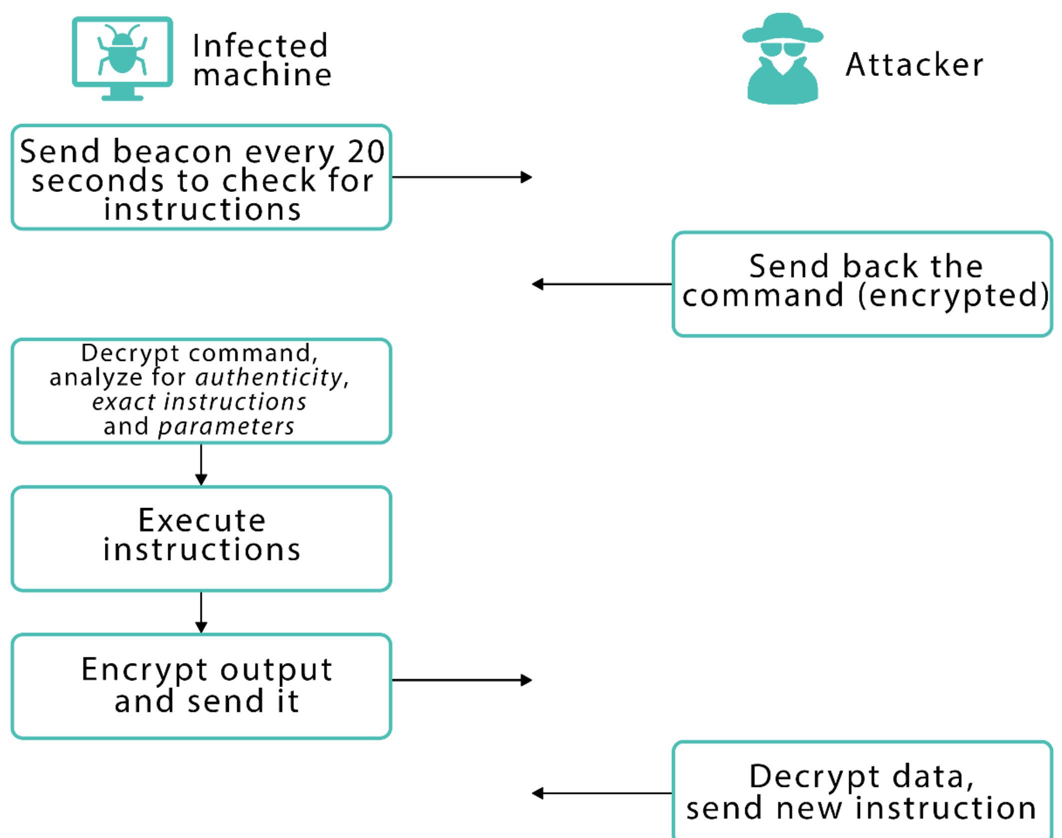


Figure 14. DarkComet communication scheme

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 Palestina. 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.

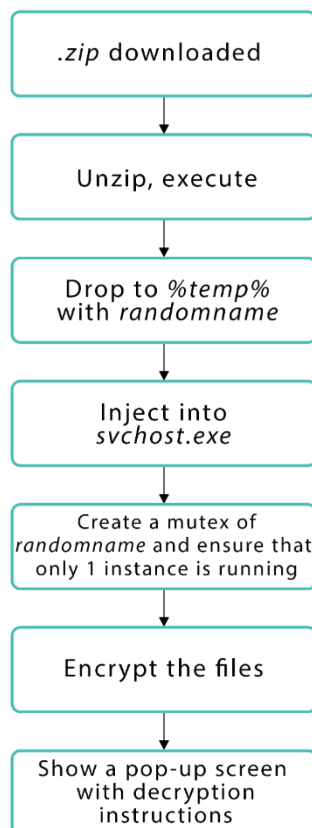


Figure 15. CTB-Locker operation

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).

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.

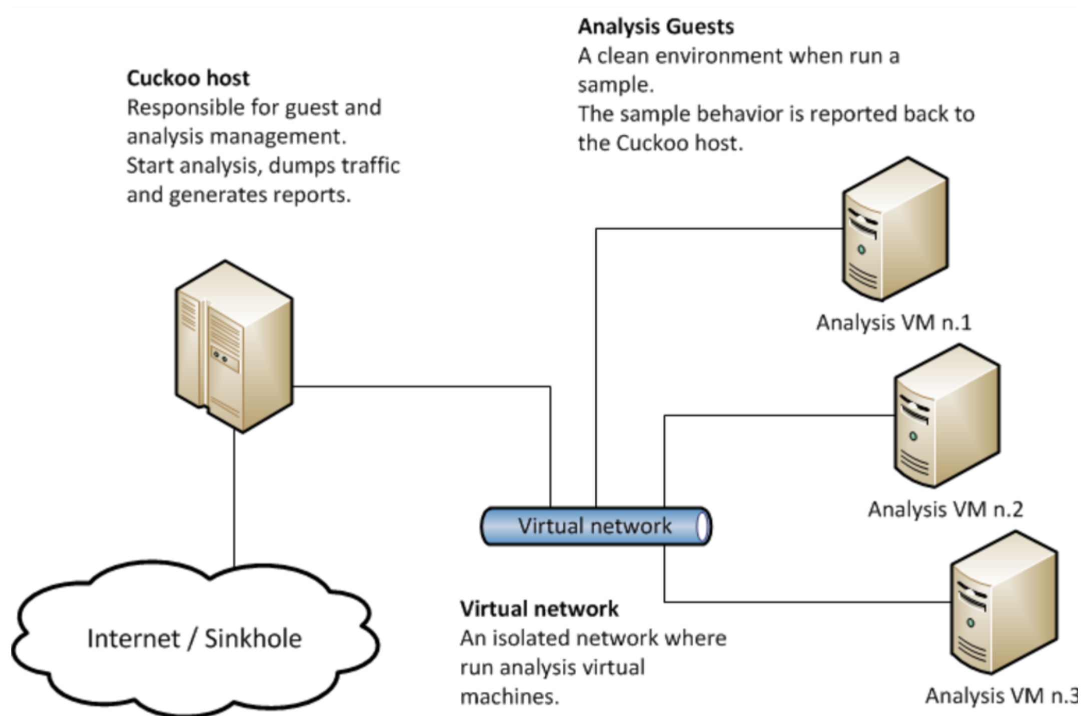


Figure 17. Cuckoo Sandbox architecture (Cuckoo Foundation 2015).

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.

Signatures
Queries for the computername (5 events)
This executable has a PDB path (1 event)
The executable has PE anomalies (could be a false positive) (1 event)
One or more processes crashed (6 events)
Checks adapter addresses which can be used to detect virtual network interfaces (1 event)
Drops a binary and executes it (1 event)
Allocates read-write-execute memory (usually to unpack itself) (2 events)
A process attempted to delay the analysis task. (1 event)
Creates a suspicious process (4 events)
Executes one or more WMI queries (1 event)
One or more potentially interesting buffers were extracted, these generally contain injected code, configuration data, etc.
Performs some HTTP requests (6 events)
Creates an Alternate Data Stream (ADS) (3 events)
Deletes its original binary from disk (1 event)
Attempts to detect Cuckoo Sandbox through the presence of a file (1 event)
Installs itself for autorun at Windows startup (1 event)
Removes the Shadow Copy to avoid recovery of the system (1 event)
One or more of the buffers contains an embedded PE file (1 event)
Executed a process and injected code into it, probably while unpacking (14 events)
File has been identified by 44 AntiVirus engines on VirusTotal as malicious (44 events)

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.

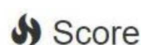
Family	Average Cuckoo score
Benign	1.04
Dridex	5.26
Locky	6.41
Teslacrypt	6.27
Vawtrak	2.66
Zeus	6.46
DarkComet	5.15
CyberGate	6.57
Xtreme	5.15
CTB-Locker	4.76

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.

104	2016-12-12 06:27	f27f3bd810aece963a520583c6481a88	teslacrypt-f27f3bd810aece963a520583c6481a88	reported	score: 3
95	2016-12-12 06:11	d7a72a833bbca58196537a6345b9f4ed	teslacrypt-d7a72a833bbca58196537a6345b9f4ed	reported	score: 7
94	2016-12-12 06:11	d6609f0e82f65102b7d33417f2e8c599	teslacrypt-d6609f0e82f65102b7d33417f2e8c599	reported	score: 9

Figure 19. Color labels of the severity of reports


Score

This file appears fairly benign with a score of **10.4 out of 10.**

Figure 20. Bugs in the Cuckoo scoring systems

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.

$$
API_{bin} = \begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_n \end{array} \begin{array}{c} \begin{array}{cccc} API_1 & API_2 & \dots & API_n \end{array} \\ \begin{bmatrix} 1 & 1 & \cdots & 0 \\ 1 & 0 & \cdots & 1 \\ \vdots & \ddots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix} \end{array}
$$

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.

$$
API_{freq} = \begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_n \end{array} \begin{array}{c} \begin{array}{cccc} API_1 & API_2 & \dots & API_n \end{array} \\ \begin{bmatrix} 112 & 312 & \cdots & 72 \\ 16 & 23 & \cdots & 315 \\ \vdots & \ddots & \ddots & \vdots \\ 157 & 1 & \cdots & 567 \end{bmatrix} \end{array}
$$

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.

$$
\begin{array}{c} \\ Combination = \end{array}
\begin{array}{c} \\ S_1 \\ S_2 \\ \vdots \\ S_n \end{array}
\begin{array}{cccccccccc}
Pass_1 & \dots & Pass_n & Fail_1 & \dots & Fail_n & RetC_1 & \dots & RetC_n \\
\end{array}
$$

$$
Combination = \begin{array}{c} S_1 \\ S_2 \\ \vdots \\ S_n \end{array}
\begin{bmatrix}
23 & \cdots & 3 & 224 & \cdots & 123 & 23 & \cdots & 27 \\
52 & \cdots & 21 & 224 & \cdots & 57 & 224 & \cdots & 1 \\
\vdots & \ddots & \ddots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
52 & \cdots & 22 & 210 & \cdots & 46 & 72 & \cdots & 111
\end{bmatrix}
$$

Here the rows represent the samples, the columns $Pass_1 \dots Pass_n$ 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 $Fail_1 \dots Fail_n$ represent a number of times each API call failed. Columns $RetC_1 \dots RetC_n$ 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
- Visual Studio redistributable packages 2005 - 2013.
- 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.

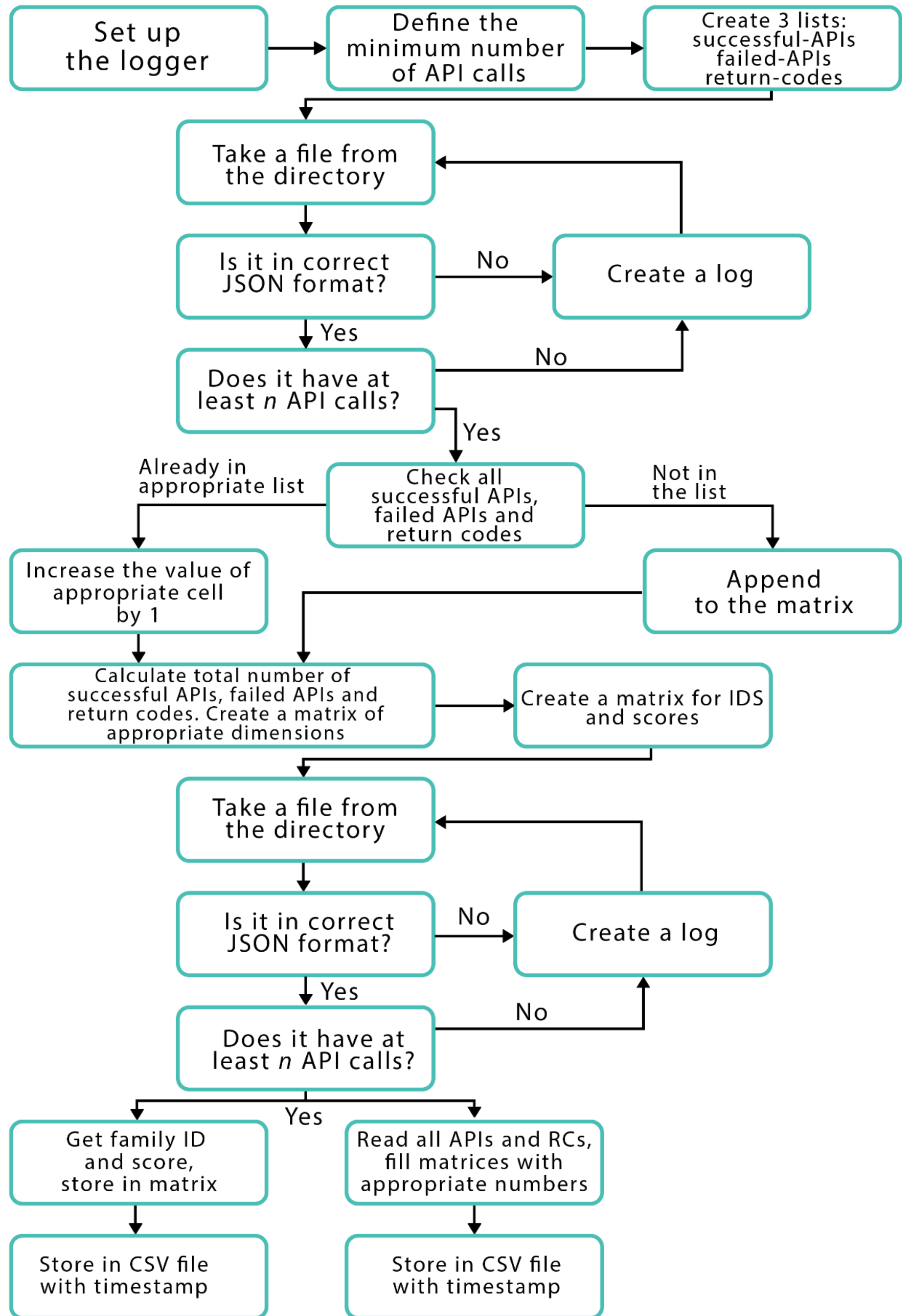


Figure 21. Feature extraction process

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 – e*1071*
- Random Forest – *randomForest*
- CrossTable plotting – *gmodels*

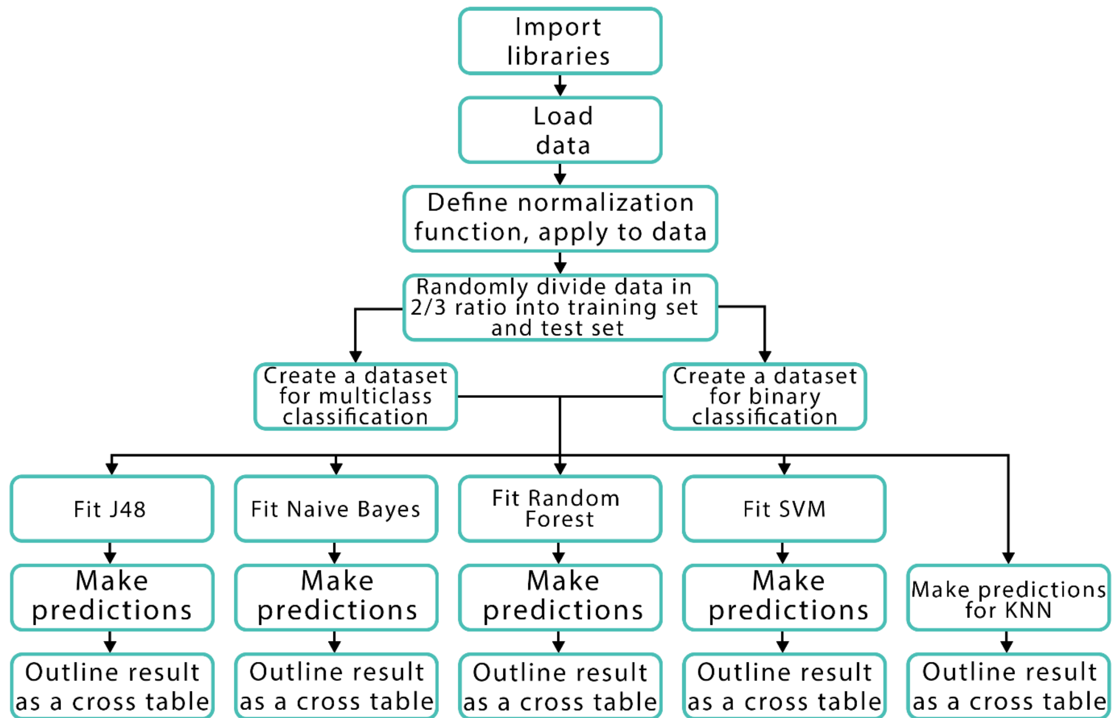


Figure 22. Machine learning classification scheme

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:

$$Accuracy = \frac{count(Correctly\ identified\ samples)}{count(Total\ samples)} \quad [15]$$

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.

```
Total Observations in Table:  371


                        | model_pred
selected_apis.testlabels |         1 |         2 |         3 |         4 |         5 |         6 |         7 |         8 |         9 |        10 | Row Total |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       0 |         0 |         0 |         1 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         1 |
                         |     0.000 |     0.000 |     1.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.003 |
                         |     0.000 |     0.000 |     0.034 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
                         |     0.000 |     0.000 |     0.003 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       1 |        49 |         0 |         1 |         0 |         0 |         0 |         2 |         2 |         5 |         2 |        61 |
                         |     0.803 |     0.000 |     0.016 |     0.000 |     0.000 |     0.000 |     0.033 |     0.033 |     0.082 |     0.033 |     0.164 |
                         |     0.860 |     0.000 |     0.034 |     0.000 |     0.000 |     0.000 |     0.038 |     0.049 |     0.132 |     0.091 |           |
                         |     0.132 |     0.000 |     0.003 |     0.000 |     0.000 |     0.000 |     0.005 |     0.005 |     0.013 |     0.005 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       2 |         1 |        31 |         2 |         0 |         1 |         2 |         0 |         0 |         0 |         0 |        37 |
                         |     0.027 |     0.838 |     0.054 |     0.000 |     0.027 |     0.054 |     0.000 |     0.000 |     0.000 |     0.000 |     0.100 |
                         |     0.018 |     0.969 |     0.069 |     0.000 |     0.045 |     0.059 |     0.000 |     0.000 |     0.000 |     0.000 |           |
                         |     0.003 |     0.084 |     0.005 |     0.000 |     0.003 |     0.005 |     0.000 |     0.000 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       3 |         2 |         0 |        22 |         1 |         0 |         0 |         0 |         0 |         0 |         2 |        27 |
                         |     0.074 |     0.000 |     0.815 |     0.037 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.074 |     0.073 |
                         |     0.035 |     0.000 |     0.759 |     0.023 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.091 |           |
                         |     0.005 |     0.000 |     0.059 |     0.003 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.005 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       4 |         0 |         0 |         0 |        43 |         1 |         0 |         0 |         0 |         0 |         0 |        44 |
                         |     0.000 |     0.000 |     0.000 |     0.977 |     0.023 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.119 |
                         |     0.000 |     0.000 |     0.000 |     0.977 |     0.045 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
                         |     0.000 |     0.000 |     0.000 |     0.116 |     0.003 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       5 |         2 |         1 |         0 |         0 |        15 |         0 |         0 |         0 |         0 |         0 |        18 |
                         |     0.111 |     0.056 |     0.000 |     0.000 |     0.833 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.049 |
                         |     0.035 |     0.031 |     0.000 |     0.000 |     0.682 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
                         |     0.005 |     0.003 |     0.000 |     0.000 |     0.040 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       6 |         3 |         0 |         0 |         0 |         4 |        30 |         0 |         1 |         1 |         1 |        40 |
                         |     0.075 |     0.000 |     0.000 |     0.000 |     0.100 |     0.750 |     0.000 |     0.025 |     0.025 |     0.025 |     0.108 |
                         |     0.053 |     0.000 |     0.000 |     0.000 |     0.182 |     0.882 |     0.000 |     0.024 |     0.026 |     0.045 |           |
                         |     0.008 |     0.000 |     0.000 |     0.000 |     0.011 |     0.081 |     0.000 |     0.003 |     0.003 |     0.003 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       7 |         0 |         0 |         1 |         0 |         0 |         0 |        47 |         0 |         1 |         0 |        49 |
                         |     0.000 |     0.000 |     0.020 |     0.000 |     0.000 |     0.000 |     0.959 |     0.000 |     0.020 |     0.000 |     0.132 |
                         |     0.000 |     0.000 |     0.034 |     0.000 |     0.000 |     0.000 |     0.904 |     0.000 |     0.026 |     0.000 |           |
                         |     0.000 |     0.000 |     0.003 |     0.000 |     0.000 |     0.000 |     0.127 |     0.000 |     0.003 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       8 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |        38 |         0 |         0 |        38 |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     1.000 |     0.000 |     0.000 |     0.102 |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.927 |     0.000 |     0.000 |           |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.102 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       9 |         0 |         0 |         0 |         0 |         0 |         1 |         2 |         0 |        31 |         0 |        34 |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.029 |     0.059 |     0.000 |     0.912 |     0.000 |     0.092 |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.029 |     0.038 |     0.000 |     0.816 |     0.000 |           |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.003 |     0.005 |     0.000 |     0.084 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                      10 |         0 |         0 |         2 |         0 |         1 |         1 |         1 |         0 |         0 |        17 |        22 |
                         |     0.000 |     0.000 |     0.091 |     0.000 |     0.045 |     0.045 |     0.045 |     0.000 |     0.000 |     0.773 |     0.059 |
                         |     0.000 |     0.000 |     0.069 |     0.000 |     0.045 |     0.029 |     0.019 |     0.000 |     0.000 |     0.773 |           |
                         |     0.000 |     0.000 |     0.005 |     0.000 |     0.003 |     0.003 |     0.003 |     0.000 |     0.000 |     0.046 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
            Column Total |        57 |        32 |        29 |        44 |        22 |        34 |        52 |        41 |        38 |        22 |       371 |
                         |     0.154 |     0.086 |     0.078 |     0.119 |     0.059 |     0.092 |     0.140 |     0.111 |     0.102 |     0.059 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
```

Figure 23. CrossTable for KNN multi-class classification

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.

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

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 traininig 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.

```
Total Observations in Table:  371

                               | model_twoway
selected_apis.testlabels.twoway |         1 |         2 | Row Total |
-------------------------------|-----------|-----------|-----------|
                             1 |        49 |        12 |        61 |
                               |     0.803 |     0.197 |     0.164 |
                               |     0.860 |     0.038 |           |
                               |     0.132 |     0.032 |           |
-------------------------------|-----------|-----------|-----------|
                             2 |         8 |       302 |       310 |
                               |     0.026 |     0.974 |     0.836 |
                               |     0.140 |     0.962 |           |
                               |     0.022 |     0.814 |           |
-------------------------------|-----------|-----------|-----------|
                  Column Total |        57 |       314 |       371 |
                               |     0.154 |     0.846 |           |
-------------------------------|-----------|-----------|-----------|
```

Figure 24. CrossTable for KNN binary classification

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.

Class	Correctly identified instances	Incorrectly identified instances	Accuracy
Benign	49	12	80.3%
Malicious	302	8	97.4%

Table 4.1. KNN binary accuracy

True positives	True negatives	False positives	False negatives
302	49	12	8

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.

```
Total Observations in Table:  371


                          | predictions 
selected_apis.testlabels  |         1 |         2 |         3 |         4 |         5 |         6 |         7 |         8 |         9 |        10 | Row Total | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                        0 |         1 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         1 | 
                          |     3.347 |     0.086 |     0.057 |     0.100 |     0.038 |     0.151 |     0.132 |     0.100 |     0.084 |     0.059 |           | 
                          |     1.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.003 | 
                          |     0.014 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
                          |     0.003 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                        1 |        56 |         0 |         0 |         0 |         0 |         5 |         0 |         0 |         0 |         0 |        61 | 
                          |   164.742 |     5.261 |     3.453 |     6.084 |     2.302 |     1.923 |     8.057 |     6.084 |     5.097 |     3.617 |           | 
                          |     0.918 |     0.000 |     0.000 |     0.000 |     0.000 |     0.082 |     0.000 |     0.000 |     0.000 |     0.000 |     0.164 | 
                          |     0.778 |     0.000 |     0.000 |     0.000 |     0.000 |     0.089 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
                          |     0.151 |     0.000 |     0.000 |     0.000 |     0.000 |     0.013 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                        2 |         3 |        32 |         0 |         0 |         1 |         1 |         0 |         0 |         0 |         0 |        37 | 
                          |     2.434 |   260.056 |     2.094 |     3.690 |     0.112 |     3.764 |     4.887 |     3.690 |     3.092 |     2.194 |           | 
                          |     0.081 |     0.865 |     0.000 |     0.000 |     0.027 |     0.027 |     0.000 |     0.000 |     0.000 |     0.000 |     0.100 | 
                          |     0.042 |     1.000 |     0.000 |     0.000 |     0.071 |     0.018 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
                          |     0.008 |     0.086 |     0.000 |     0.000 |     0.003 |     0.003 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                        3 |         2 |         0 |        21 |         0 |         0 |         4 |         0 |         0 |         0 |         0 |        27 | 
                          |     2.003 |     2.329 |   248.084 |     2.693 |     1.019 |     0.001 |     3.566 |     2.693 |     2.256 |     1.601 |           | 
                          |     0.074 |     0.000 |     0.778 |     0.000 |     0.000 |     0.148 |     0.000 |     0.000 |     0.000 |     0.000 |     0.073 | 
                          |     0.028 |     0.000 |     1.000 |     0.000 |     0.000 |     0.071 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
                          |     0.005 |     0.000 |     0.057 |     0.000 |     0.000 |     0.011 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                        4 |         0 |         0 |         0 |        37 |         1 |         6 |         0 |         0 |         0 |         0 |        44 | 
                          |     8.539 |     3.795 |     2.491 |   242.365 |     0.263 |     0.062 |     5.811 |     4.388 |     3.677 |     2.609 |           | 
                          |     0.000 |     0.000 |     0.000 |     0.841 |     0.023 |     0.136 |     0.000 |     0.000 |     0.000 |     0.000 |     0.119 | 
                          |     0.000 |     0.000 |     0.000 |     1.000 |     0.071 |     0.107 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
                          |     0.000 |     0.000 |     0.000 |     0.100 |     0.003 |     0.016 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                        5 |         3 |         0 |         0 |         0 |        10 |         5 |         0 |         0 |         0 |         0 |        18 | 
                          |     0.070 |     1.553 |     1.019 |     1.795 |   127.901 |     1.918 |     2.377 |     1.795 |     1.504 |     1.067 |           | 
                          |     0.167 |     0.000 |     0.000 |     0.000 |     0.556 |     0.278 |     0.000 |     0.000 |     0.000 |     0.000 |     0.049 | 
                          |     0.042 |     0.000 |     0.000 |     0.000 |     0.714 |     0.089 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
                          |     0.008 |     0.000 |     0.000 |     0.000 |     0.027 |     0.013 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                        6 |         7 |         0 |         0 |         0 |         2 |        31 |         0 |         0 |         0 |         0 |        40 | 
                          |     0.075 |     3.450 |     2.264 |     3.989 |     0.159 |   103.203 |     5.283 |     3.989 |     3.342 |     2.372 |           | 
                          |     0.175 |     0.000 |     0.000 |     0.000 |     0.050 |     0.775 |     0.000 |     0.000 |     0.000 |     0.000 |     0.108 | 
                          |     0.097 |     0.000 |     0.000 |     0.000 |     0.143 |     0.554 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
                          |     0.019 |     0.000 |     0.000 |     0.000 |     0.005 |     0.084 |     0.000 |     0.000 |     0.000 |     0.000 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                        7 |         0 |         0 |         0 |         0 |         0 |         1 |        48 |         0 |         0 |         0 |        49 | 
                          |     9.509 |     4.226 |     2.774 |     4.887 |     1.849 |     5.531 |   266.483 |     4.887 |     4.094 |     2.906 |           | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.020 |     0.980 |     0.000 |     0.000 |     0.000 |     0.132 | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.018 |     0.980 |     0.000 |     0.000 |     0.000 |           | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.003 |     0.129 |     0.000 |     0.000 |     0.000 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                        8 |         0 |         0 |         0 |         0 |         0 |         0 |         1 |        37 |         0 |         0 |        38 | 
                          |     7.375 |     3.278 |     2.151 |     3.790 |     1.434 |     5.736 |     3.218 |   291.027 |     3.175 |     2.253 |           | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.026 |     0.974 |     0.000 |     0.000 |     0.102 | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.020 |     1.000 |     0.000 |     0.000 |           | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.003 |     0.100 |     0.000 |     0.000 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                        9 |         0 |         0 |         0 |         0 |         0 |         3 |         0 |         0 |        31 |         0 |        34 | 
                          |     6.598 |     2.933 |     1.925 |     3.391 |     1.283 |     0.886 |     4.491 |     3.391 |   279.106 |     2.016 |           | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.088 |     0.000 |     0.000 |     0.912 |     0.000 |     0.092 | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.054 |     0.000 |     0.000 |     1.000 |     0.000 |           | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.008 |     0.000 |     0.000 |     0.084 |     0.000 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       10 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |        22 |        22 | 
                          |     4.270 |     1.898 |     1.245 |     2.194 |     0.830 |     3.321 |     2.906 |     2.194 |     1.838 |   328.305 |           | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     1.000 |     0.059 | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     1.000 |           | 
                          |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.059 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
             Column Total |        72 |        32 |        21 |        37 |        14 |        56 |        49 |        37 |        31 |        22 |       371 | 
                          |     0.194 |     0.086 |     0.057 |     0.100 |     0.038 |     0.151 |     0.132 |     0.100 |     0.084 |     0.059 |           | 
--------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
```

Figure 25. SVM CrossTable

The detailed information about the accuracy of each class can be found in Table 5.

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

Table 5. SVM multiclass accuracy

```
   Cell Contents
|-------------------------|
|                       N |
| Chi-square contribution |
|           N / Row Total |
|           N / Col Total |
|         N / Table Total |
|-------------------------|


Total Observations in Table:  371


                               | predictions 
selected_apis.testlabels.twoway |         1 |         2 | Row Total | 
-------------------------------|-----------|-----------|-----------|
                             1 |        41 |        20 |        61 | 
                               |   174.102 |    21.631 |           | 
                               |     0.672 |     0.328 |     0.164 | 
                               |     1.000 |     0.061 |           | 
                               |     0.111 |     0.054 |           | 
-------------------------------|-----------|-----------|-----------|
                             2 |         0 |       310 |       310 | 
                               |    34.259 |     4.256 |           | 
                               |     0.000 |     1.000 |     0.836 | 
                               |     0.000 |     0.939 |           | 
                               |     0.000 |     0.836 |           | 
-------------------------------|-----------|-----------|-----------|
                  Column Total |        41 |       330 |       371 | 
                               |     0.111 |     0.889 |           | 
-------------------------------|-----------|-----------|-----------|
```

Figure 26. SVM binary classification CrossTable

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.

Class	Correctly identified instances	Incorrectly identified instances	Accuracy
Benign	41	20	67.2%
Malicious	310	0	100%

Table 6.1. SVM binary classification accuracy

True positives	True negatives	False positives	False negatives
310	41	20	0

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.

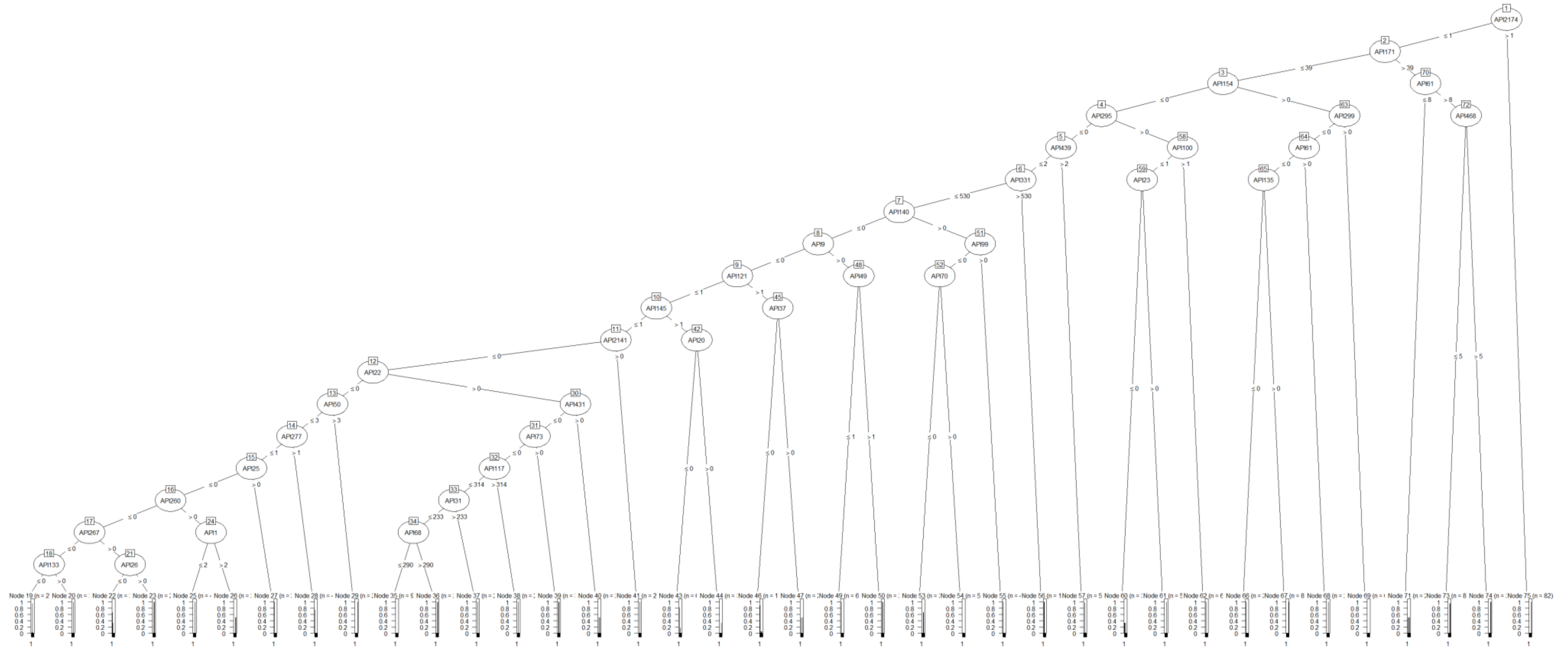


Figure 27. Multiclass Decision Tree

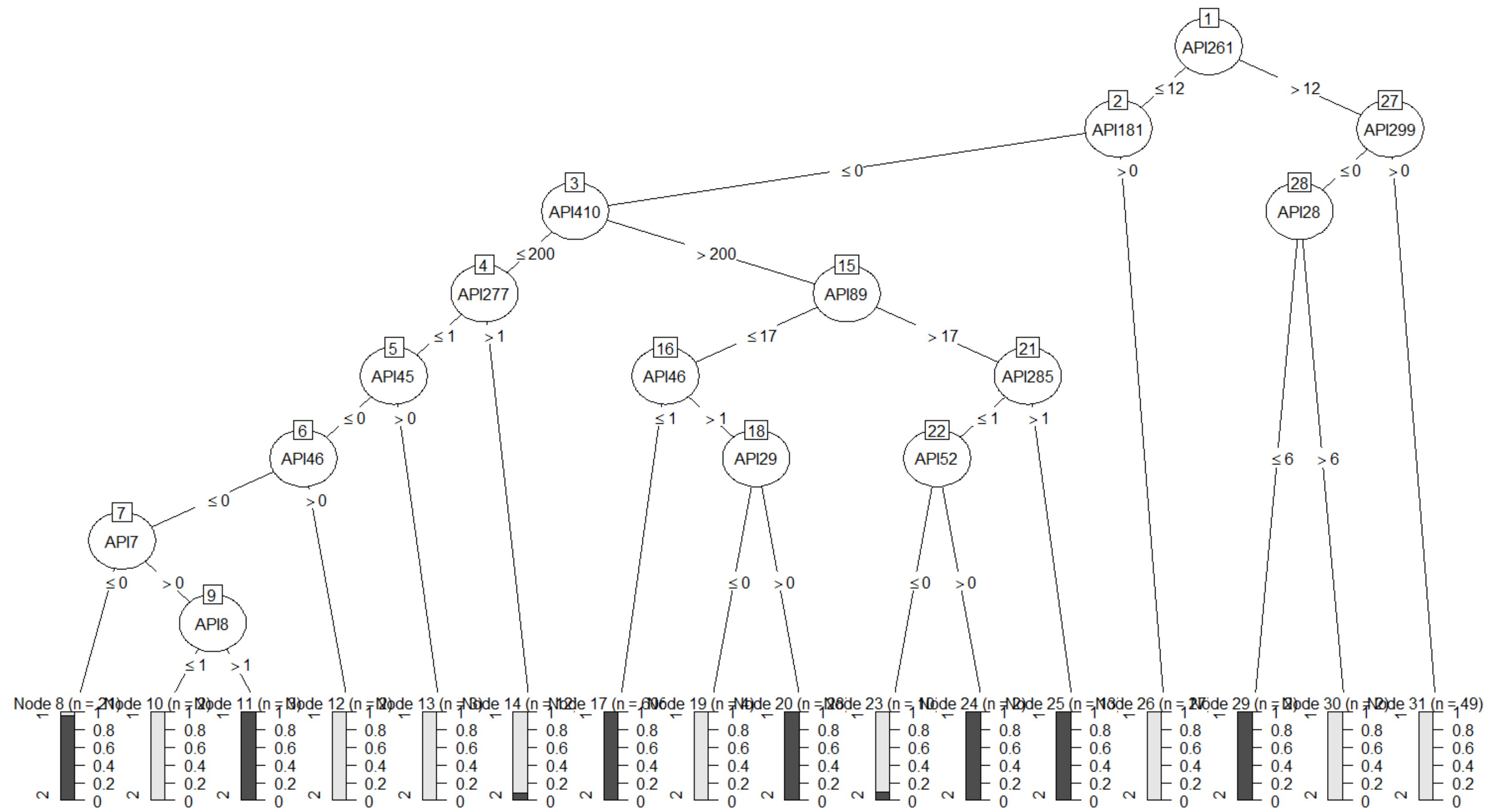


Figure 28. Binary Decision Tree

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.

Total Observations in Table: 371

selected_apis.testlabels	predictions 1	2	3	4	5	6	7	8	9	10	Row Total
0	1 4.447 1.000 0.017 0.003	0 0.111 0.000 0.000 0.000	0 0.065 0.000 0.000 0.000	0 0.121 0.000 0.000 0.000	0 0.059 0.000 0.000 0.000	0 0.100 0.000 0.000 0.000	0 0.127 0.000 0.000 0.000	0 0.111 0.000 0.000 0.000	0 0.089 0.000 0.000 0.000	0 0.059 0.000 0.000 0.000	1 0.003
1	54 202.294 0.885 0.915 0.146	0 6.741 0.000 0.000 0.000	0 3.946 0.000 0.000 0.000	1 5.534 0.016 0.022 0.003	4 0.040 0.066 0.182 0.011	1 4.248 0.016 0.027 0.003	0 7.728 0.000 0.000 0.000	0 6.741 0.000 0.000 0.000	1 3.610 0.016 0.030 0.003	0 3.617 0.000 0.000 0.000	61 0.164
2	0 5.884 0.000 0.000 0.000	37 264.894 1.000 0.902 0.100	0 2.394 0.000 0.000 0.000	0 4.488 0.000 0.000 0.000	0 2.194 0.000 0.000 0.000	0 3.690 0.000 0.000 0.000	0 4.687 0.000 0.000 0.000	0 4.089 0.000 0.000 0.000	0 3.291 0.000 0.000 0.000	0 2.194 0.000 0.000 0.000	37 0.100
3	2 1.225 0.074 0.034 0.005	0 2.984 0.000 0.000 0.000	24 283.524 0.889 1.000 0.065	0 3.275 0.000 0.000 0.000	0 1.601 0.000 0.000 0.000	0 2.693 0.000 0.000 0.000	0 3.420 0.000 0.000 0.000	0 2.984 0.000 0.000 0.000	0 2.402 0.000 0.000 0.000	1 0.226 0.037 0.045 0.003	27 0.073
4	0 6.997 0.000 0.000 0.000	0 4.863 0.000 0.000 0.000	0 2.846 0.000 0.000 0.000	44 280.092 1.000 0.978 0.119	0 2.609 0.000 0.000 0.000	0 4.388 0.000 0.000 0.000	0 5.574 0.000 0.000 0.000	0 4.863 0.000 0.000 0.000	0 3.914 0.000 0.000 0.000	0 2.609 0.000 0.000 0.000	44 0.119
5	1 1.212 0.056 0.017 0.003	1 0.492 0.056 0.024 0.003	0 1.164 0.000 0.000 0.000	0 2.183 0.000 0.000 0.000	16 208.906 0.889 0.727 0.043	0 1.795 0.000 0.000 0.000	0 2.280 0.000 0.000 0.000	0 1.989 0.000 0.000 0.000	0 1.601 0.000 0.000 0.000	0 1.067 0.000 0.000 0.000	18 0.049
6	1 4.518 0.025 0.017 0.003	3 0.456 0.075 0.073 0.008	0 2.588 0.000 0.000 0.000	0 4.852 0.000 0.000 0.000	2 0.058 0.050 0.091 0.005	33 210.975 0.825 0.892 0.089	0 5.067 0.000 0.000 0.000	1 2.647 0.025 0.024 0.003	0 3.558 0.000 0.000 0.000	0 2.372 0.000 0.000 0.000	40 0.108
7	0 7.792 0.000 0.000 0.000	0 5.415 0.000 0.000 0.000	0 3.170 0.000 0.000 0.000	0 5.943 0.000 0.000 0.000	0 2.906 0.000 0.000 0.000	0 4.887 0.000 0.000 0.000	47 268.065 0.959 1.000 0.127	2 2.154 0.041 0.049 0.005	0 4.358 0.000 0.000 0.000	0 2.906 0.000 0.000 0.000	49 0.132
8	0 6.043 0.000 0.000 0.000	0 4.199 0.000 0.000 0.000	0 2.458 0.000 0.000 0.000	0 4.609 0.000 0.000 0.000	0 2.253 0.000 0.000 0.000	0 3.790 0.000 0.000 0.000	0 4.814 0.000 0.000 0.000	38 272.053 1.000 0.927 0.102	0 3.380 0.000 0.000 0.000	0 2.253 0.000 0.000 0.000	38 0.102
9	0 5.407 0.000 0.000 0.000	0 3.757 0.000 0.000 0.000	0 2.199 0.000 0.000 0.000	0 4.124 0.000 0.000 0.000	0 2.016 0.000 0.000 0.000	2 0.570 0.059 0.054 0.005	0 4.307 0.000 0.000 0.000	0 3.757 0.000 0.000 0.000	32 277.620 0.941 0.970 0.086	0 2.016 0.000 0.000 0.000	34 0.092
10	0 3.499 0.000 0.000 0.000	0 2.431 0.000 0.000 0.000	0 1.423 0.000 0.000 0.000	0 2.668 0.000 0.000 0.000	0 1.305 0.000 0.000 0.000	1 0.650 0.045 0.027 0.003	0 2.787 0.000 0.000 0.000	0 2.431 0.000 0.000 0.000	0 1.957 0.000 0.000 0.000	21 297.344 0.955 0.955 0.057	22 0.059
Column Total	59 0.159	41 0.111	24 0.065	45 0.121	22 0.059	37 0.100	47 0.127	41 0.111	33 0.089	22 0.059	371

Figure 29. Decision Tree multi-class CrossTable

The detailed results of each malware family can be found in Table 7.

Class	Family	Correctly classified	Incorrectly classified	Accuracy	Average Cuckoo score
1	Benign	54	7	88.5%	1.04
2	Dridex	37	0	100%	5.26
3	Locky	24	3	88.9%	6.41
4	TeslaCrypt	44	0	100%	6.27
5	Vawtrak	16	2	88.9%	2.66
6	Zeus	33	7	82.5%	6.46
7	DarkComet	47	2	95.9%	5.15
8	CyberGate	38	0	100%	6.57
9	Xtreme	32	2	94.1%	5.15
10	CTB-Locker	21	1	95.5%	4.76

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.

```
                                | predictions
selected_apis.testlabels.twoway |          1 |          2 |  Row Total |
--------------------------------|------------|------------|------------|
                              1 |         46 |         15 |         61 |
                                |    168.727 |     26.891 |            |
                                |      0.754 |      0.246 |      0.164 |
                                |      0.902 |      0.047 |            |
                                |      0.124 |      0.040 |            |
--------------------------------|------------|------------|------------|
                              2 |          5 |        305 |        310 |
                                |     33.201 |      5.291 |            |
                                |      0.016 |      0.984 |      0.836 |
                                |      0.098 |      0.953 |            |
                                |      0.013 |      0.822 |            |
--------------------------------|------------|------------|------------|
                   Column Total |         51 |        320 |        371 |
                                |      0.137 |      0.863 |            |
--------------------------------|------------|------------|------------|
```

Figure 30. Decision Tree binary classification CrossTable

Class	Correctly identified instances	Incorrectly identified instances	Accuracy
Benign	46	15	75.4%
Malicious	305	5	98.4%

Table 8.1. Decision Tree binary classification accuracy

True positives	True negatives	False positives	False negatives
305	46	15	5

Table 8.2. 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.

selected_apis.testlabels	predictions 1	2	3	4	5	6	7	8	9	10	Row Total
0	0 0.094 0.000 0.000 0.000	0 0.005 0.000 0.000 0.000	0 0.084 0.000 0.000 0.000	0 0.102 0.000 0.000 0.000	0 0.054 0.000 0.000 0.000	0 0.075 0.000 0.000 0.000	1 4.662 1.000 0.018 0.003	0 0.100 0.000 0.000 0.000	0 0.127 0.000 0.000 0.000	0 0.205 0.000 0.000 0.000	1 0.003
1	34 138.633 0.557 0.971 0.092	0 0.329 0.000 0.000 0.000	2 1.882 0.033 0.065 0.005	0 6.248 0.000 0.000 0.000	1 1.593 0.016 0.050 0.003	0 4.604 0.000 0.000 0.000	2 5.799 0.033 0.035 0.005	0 6.084 0.000 0.000 0.000	4 1.798 0.066 0.085 0.011	18 2.424 0.295 0.237 0.049	61 0.164
2	0 3.491 0.000 0.000 0.000	1 3.213 0.027 0.500 0.003	4 0.267 0.108 0.129 0.011	0 3.790 0.000 0.000 0.000	11 40.658 0.297 0.550 0.030	0 2.792 0.000 0.000 0.000	0 5.685 0.000 0.000 0.000	0 3.690 0.000 0.000 0.000	3 0.607 0.081 0.064 0.008	18 14.326 0.486 0.237 0.049	37 0.100
3	0 2.547 0.000 0.000 0.000	0 0.146 0.000 0.000 0.000	25 229.287 0.926 0.806 0.067	0 2.765 0.000 0.000 0.000	0 1.456 0.000 0.000 0.000	0 2.038 0.000 0.000 0.000	0 4.148 0.000 0.000 0.000	0 2.693 0.000 0.000 0.000	0 3.420 0.000 0.000 0.000	2 2.254 0.074 0.026 0.005	27 0.073
4	0 4.151 0.000 0.000 0.000	0 0.237 0.000 0.000 0.000	0 3.677 0.000 0.000 0.000	33 180.145 0.750 0.868 0.089	0 2.372 0.000 0.000 0.000	0 3.321 0.000 0.000 0.000	0 6.760 0.000 0.000 0.000	0 4.388 0.000 0.000 0.000	8 1.056 0.182 0.170 0.022	3 4.012 0.068 0.039 0.008	44 0.119
5	0 1.698 0.000 0.000 0.000	0 0.097 0.000 0.000 0.000	0 1.504 0.000 0.000 0.000	2 0.013 0.111 0.053 0.005	8 50.926 0.444 0.400 0.022	0 1.358 0.000 0.000 0.000	1 1.127 0.056 0.018 0.003	0 1.795 0.000 0.000 0.000	0 2.280 0.000 0.000 0.000	7 2.976 0.389 0.092 0.019	18 0.049
6	0 3.774 0.000 0.000 0.000	0 0.216 0.000 0.000 0.000	0 3.342 0.000 0.000 0.000	3 0.294 0.075 0.079 0.008	0 2.156 0.000 0.000 0.000	28 206.719 0.700 1.000 0.075	2 2.796 0.050 0.035 0.005	0 3.989 0.000 0.000 0.000	1 3.265 0.025 0.021 0.003	6 0.587 0.150 0.079 0.016	40 0.108
7	0 4.623 0.000 0.000 0.000	0 0.264 0.000 0.000 0.000	0 4.094 0.000 0.000 0.000	0 5.019 0.000 0.000 0.000	0 2.642 0.000 0.000 0.000	0 3.698 0.000 0.000 0.000	49 228.458 1.000 0.860 0.132	0 4.887 0.000 0.000 0.000	0 6.208 0.000 0.000 0.000	0 10.038 0.000 0.000 0.000	49 0.132
8	0 3.585 0.000 0.000 0.000	0 0.205 0.000 0.000 0.000	0 3.175 0.000 0.000 0.000	0 3.892 0.000 0.000 0.000	0 2.049 0.000 0.000 0.000	0 2.868 0.000 0.000 0.000	1 4.010 0.026 0.018 0.003	37 291.027 0.974 1.000 0.100	0 4.814 0.000 0.000 0.000	0 7.784 0.000 0.000 0.000	38 0.102
9	1 1.519 0.029 0.029 0.003	1 3.639 0.029 0.500 0.003	0 2.841 0.000 0.000 0.000	0 3.482 0.000 0.000 0.000	0 1.833 0.000 0.000 0.000	0 2.566 0.000 0.000 0.000	1 3.415 0.029 0.018 0.003	0 3.391 0.000 0.000 0.000	31 165.418 0.912 0.660 0.084	0 6.965 0.000 0.000 0.000	34 0.092
10	0 2.075 0.000 0.000 0.000	0 0.119 0.000 0.000 0.000	0 1.838 0.000 0.000 0.000	0 2.253 0.000 0.000 0.000	0 1.186 0.000 0.000 0.000	0 1.660 0.000 0.000 0.000	0 3.380 0.000 0.000 0.000	0 2.194 0.000 0.000 0.000	0 2.787 0.000 0.000 0.000	22 67.901 1.000 0.289 0.059	22 0.059
Column Total	35 0.094	2 0.005	31 0.084	38 0.102	20 0.054	28 0.075	57 0.154	37 0.100	47 0.127	76 0.205	371

Figure 31. Naive Bayes multi-class classification cross-table

The detailed results that outline the accuracy of each of the malware families can be found in Table 9.

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

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.

```
Total Observations in Table:  371

                               | predictions
selected_apis.testlabels.twoway |         1 |         2 | Row Total |
-------------------------------|-----------|-----------|-----------|
                             1 |        61 |         0 |        61 |
                               |    14.747 |    23.512 |           |
                               |     1.000 |     0.000 |     0.164 |
                               |     0.268 |     0.000 |           |
                               |     0.164 |     0.000 |           |
-------------------------------|-----------|-----------|-----------|
                             2 |       167 |       143 |       310 |
                               |     2.902 |     4.627 |           |
                               |     0.539 |     0.461 |     0.836 |
                               |     0.732 |     1.000 |           |
                               |     0.450 |     0.385 |           |
-------------------------------|-----------|-----------|-----------|
                  Column Total |       228 |       143 |       371 |
                               |     0.615 |     0.385 |           |
-------------------------------|-----------|-----------|-----------|
```

Figure 32. Naive Bayes binary classification cross-table

Class	Correctly identified instances	Incorrectly identified instances	Accuracy
Benign	61	0	100%
Malicious	143	167	46.1%

Table 10.1. Naive Bayes binary classification accuracy

True positives	True negatives	False positives	False negatives
143	61	0	167

Table 10.2. Naive Bayes binary classification accuracy

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.

```
                        | predictions
selected_apis.testlabels |         1 |         2 |         3 |         4 |         5 |         6 |         7 |         8 |         9 |        10 | Row Total |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       0 |         1 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         1 |
                         |     4.151 |     0.097 |     0.070 |     0.119 |     0.059 |     0.100 |     0.132 |     0.102 |     0.092 |     0.062 |           |
                         |     1.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.003 |
                         |     0.016 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
                         |     0.003 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       1 |        58 |         1 |         0 |         0 |         2 |         0 |         0 |         0 |         0 |         0 |        61 |
                         |   224.190 |     4.088 |     4.275 |     7.235 |     0.723 |     6.084 |     8.057 |     6.248 |     5.590 |     3.782 |           |
                         |     0.951 |     0.016 |     0.000 |     0.000 |     0.033 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.164 |
                         |     0.935 |     0.028 |     0.000 |     0.000 |     0.091 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
                         |     0.156 |     0.003 |     0.000 |     0.000 |     0.005 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       2 |         0 |        35 |         0 |         0 |         1 |         0 |         0 |         0 |         0 |         1 |        37 |
                         |     6.183 |   274.788 |     2.593 |     4.388 |     0.650 |     3.690 |     4.887 |     3.790 |     3.391 |     0.730 |           |
                         |     0.000 |     0.946 |     0.000 |     0.000 |     0.027 |     0.000 |     0.000 |     0.000 |     0.000 |     0.027 |     0.100 |
                         |     0.000 |     0.972 |     0.000 |     0.000 |     0.045 |     0.000 |     0.000 |     0.000 |     0.000 |     0.043 |           |
                         |     0.000 |     0.094 |     0.000 |     0.000 |     0.003 |     0.000 |     0.000 |     0.000 |     0.000 |     0.003 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       3 |         1 |         0 |        25 |         0 |         1 |         0 |         0 |         0 |         0 |         0 |        27 |
                         |     2.734 |     2.620 |   282.198 |     3.202 |     0.226 |     2.693 |     3.566 |     2.765 |     2.474 |     1.674 |           |
                         |     0.037 |     0.000 |     0.926 |     0.000 |     0.037 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.073 |
                         |     0.016 |     0.000 |     0.962 |     0.000 |     0.045 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
                         |     0.003 |     0.000 |     0.067 |     0.000 |     0.003 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       4 |         0 |         0 |         0 |        44 |         0 |         0 |         0 |         0 |         0 |         0 |        44 |
                         |     7.353 |     4.270 |     3.084 |   288.218 |     2.609 |     4.388 |     5.811 |     4.507 |     4.032 |     2.728 |           |
                         |     0.000 |     0.000 |     0.000 |     1.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.119 |
                         |     0.000 |     0.000 |     0.000 |     1.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
                         |     0.000 |     0.000 |     0.000 |     0.119 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       5 |         1 |         0 |         0 |         0 |        15 |         2 |         0 |         0 |         0 |         0 |        18 |
                         |     1.341 |     1.747 |     1.261 |     2.135 |   181.863 |     0.023 |     2.377 |     1.844 |     1.650 |     1.116 |           |
                         |     0.056 |     0.000 |     0.000 |     0.000 |     0.833 |     0.111 |     0.000 |     0.000 |     0.000 |     0.000 |     0.049 |
                         |     0.016 |     0.000 |     0.000 |     0.000 |     0.682 |     0.054 |     0.000 |     0.000 |     0.000 |     0.000 |           |
                         |     0.003 |     0.000 |     0.000 |     0.000 |     0.040 |     0.005 |     0.000 |     0.000 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       6 |         1 |         0 |         1 |         0 |         3 |        35 |         0 |         0 |         0 |         0 |        40 |
                         |     4.834 |     3.881 |     1.160 |     4.744 |     0.166 |   241.067 |     5.283 |     4.097 |     3.666 |     2.480 |           |
                         |     0.025 |     0.000 |     0.025 |     0.000 |     0.075 |     0.875 |     0.000 |     0.000 |     0.000 |     0.000 |     0.108 |
                         |     0.016 |     0.000 |     0.038 |     0.000 |     0.136 |     0.946 |     0.000 |     0.000 |     0.000 |     0.000 |           |
                         |     0.003 |     0.000 |     0.003 |     0.000 |     0.008 |     0.094 |     0.000 |     0.000 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       7 |         0 |         0 |         0 |         0 |         0 |         0 |        49 |         0 |         0 |         0 |        49 |
                         |     8.189 |     4.755 |     3.434 |     5.811 |     2.906 |     4.887 |   279.472 |     5.019 |     4.491 |     3.038 |           |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     1.000 |     0.000 |     0.000 |     0.000 |     0.132 |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     1.000 |     0.000 |     0.000 |     0.000 |           |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.132 |     0.000 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       8 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |        38 |         0 |         0 |        38 |
                         |     6.350 |     3.687 |     2.663 |     4.507 |     2.253 |     3.790 |     5.019 |   298.892 |     3.482 |     2.356 |           |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     1.000 |     0.000 |     0.000 |     0.102 |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     1.000 |     0.000 |     0.000 |           |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.102 |     0.000 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                       9 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |        34 |         0 |        34 |
                         |     5.682 |     3.299 |     2.383 |     4.032 |     2.016 |     3.391 |     4.491 |     3.482 |   306.116 |     2.108 |           |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     1.000 |     0.000 |     0.092 |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     1.000 |     0.000 |           |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.092 |     0.000 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
                      10 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |         0 |        22 |        22 |
                         |     3.677 |     2.135 |     1.542 |     2.609 |     1.305 |     2.194 |     2.906 |     2.253 |     2.016 |   312.233 |           |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     1.000 |     0.059 |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.957 |           |
                         |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.000 |     0.059 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
            Column Total |        62 |        36 |        26 |        44 |        22 |        37 |        49 |        38 |        34 |        23 |       371 |
                         |     0.167 |     0.097 |     0.070 |     0.119 |     0.059 |     0.100 |     0.132 |     0.102 |     0.092 |     0.062 |           |
-------------------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
```

Figure 33. Random Forest multiclass classification cross-table

The detailed information about the performance of each class can be found in Table 11.

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

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.

```
Total Observations in Table:  371

                              | predictions
selected_apis.testlabels.twoway |         1 |         2 | Row Total |
-------------------------------|-----------|-----------|-----------|
                             1 |        52 |         9 |        61 |
                               |   204.055 |    35.516 |           |
                               |     0.852 |     0.148 |     0.164 |
                               |     0.945 |     0.028 |           |
                               |     0.140 |     0.024 |           |
-------------------------------|-----------|-----------|-----------|
                             2 |         3 |       307 |       310 |
                               |    40.153 |     6.989 |           |
                               |     0.010 |     0.990 |     0.836 |
                               |     0.055 |     0.972 |           |
                               |     0.008 |     0.827 |           |
-------------------------------|-----------|-----------|-----------|
                  Column Total |        55 |       316 |       371 |
                               |     0.148 |     0.852 |           |
-------------------------------|-----------|-----------|-----------|
```

Figure 34. Random Forest binary classification cross-table

Class	Correctly identified instances	Incorrectly identified instances	Accuracy
Benign	52	9	85.2%
Malicious	307	3	99%

Table 12.1. Random Forest binary classification accuracy

True positives	True negatives	False positives	False negatives
307	52	9	3

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.

6 CONCLUSIONS

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.

Performance \ Classifier		KNN	SVM	Naive Bayes	J48	Random Forest
Multi-class	Accuracy	87%	87.6%	72.34%	93.3%	95.69%
Binary	Accuracy	94.6%	94.6%	55%	94.6%	96.8%
	False-positives	12	20	0	15	9
	False-negatives	8	0	167	5	3
	True-positives	302	310	143	305	307
	True-negatives	49	41	61	46	52

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

Alazab, Mamoun, Sitalakshmi Venkatraman, Paul Watters, and Moutaz Alazab. 2011. Zero-day Malware Detection based on Supervised Learning Algorithms of API call Signatures. *Proceedings of the 9-th Australasian Data Mining Conference,* 171-181.

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

Aquino, Maharlito. 2014. Fake BACS Remittance Emails Delivers Dridex Malware. WWW document. Available at: https://blog.cyren.com/articles/fake-bacs-remittance-emails-delivers-dridex-malware.html. [Accessed 15 February 2017].

Aziz, Naman. 2014. Cybergate RAT Tutorial For Beginners. WWW document. Available at: http://rattut.blogspot.fi/. [Accessed 15 February 2017].

Baldangombo, Usukhbayar, Nyamjav Jambaljav, and Shi-Jinn Horng. 2013. A Static Malware Detection System Using Data Mining Methods. Cornell University

Baskaran, Balaji, and Anca Ralescu. 2016. A Study of Android Malware Detection Techniques and Machine Learning. *Proceedings of the 27th Modern Artificial Intelligence and Cognitive Science Conference,* 15-23.

Biau, G´erard. 2013. Analysis of a Random Forests Model. *Journal of Machine Learning Research,* 1063-1095.

Bishop, Christopher. 2006. Pattern Recognition and Machine Learning. New York: Springer

Bremer, Jurriaan. 2015. Welcome to VMCloak's documentation! WWW document. Available at: http://vmcloak.readthedocs.io/en/latest/. [Accessed 15 February 2017]

Chien, Eric. 2005. Techniques of Adware and Spyware. WWW document. Available at: https://www.symantec.com/avcenter/reference/techniques.of .adware.and.spyware.pdf. [Accessed 15 February 2017]

Chuvakin, Anton. 2003. An Overview of Unix Rootkits. *iDEFENCE Labs*

Cuckoo Foundation. 2015. Cuckoo Sandbox Book. WWW document. Available at: http://docs.cuckoosandbox.org/en/latest/introduction/what/. [Accessed 15 February 2017]

Egele, Manuel, Theodoor Sholte, Engin Kirda, and Christofer Kruegel. 2012. A Survey on Automated Dynamic Malware Analysis Techniques and Tools. *ACM Computing Surveys*.

Falliere, Nicolas, and Eric Chien. 2009. Zeus: King of the Bots. *Symantec Corporation.*

Gavrilut, Dragos, Mihai Cimpoesu, Dan Anton, and Liviu Ciortuz. 2009. Malware Detection Using Machine Learning. *Proceedings of the International Multiconference on Computer Science and Information Technology,* 735-741.

Guyon, Isabelle, and Andre Elisseef. 2006. An Introduction to Feature Extraction. *Feature Extraction, Foundations and Applications*. New York: Springer.

Harley, David, and Andrew Lee. 2009. Heuristic Analysis — Detecting Unknown Viruses.

Horton, Jeffrey, and Jennifer Seberry. 1997. Computer Viruses. An Introduction. University of Wollongong.

Hung, Pham Van. 2011. An approach to fast malware classification with machine learning technique.

Jing, Ranzhe, and Yong Zhang. 2010. A View of Support Vector Machines Algorithm on Classification Problems. *International Conference on Multimedia Communications.*

Juniper Research. 2016. Cybercrime will cost businesses over $2 trillion by 2019. WWW document. Available at: https://www.juniperresearch.com/press/press-releases/cybercrime-cost-businesses-over-2trillion. [Accessed on 15 February 2017]

Kaspersky Lab. 2016. Kaspersky Security Bulletin 2015. Overall statistics for 2015. WWW document. Available at:https://securelist.com/analysis/kaspersky-security-bulletin/73038/kaspersky-security-bulletin-2015-overall-statistics-for-2015/. [Accessed 15 February 2017]

Kaspersky Labs. 2017. What is malware and how to defend against it? WWW document. Available at: http://usa.kaspersky.com/internet-security center/internet-safety/what-is-malware-and-how-to-protect-against-it#.WJZS9xt942x. [Accessed 15 February 2017]

Křoustek, Jakub. 2015. Analysis of Banking Trojan Vawtrak. *AVG Technologies, Virus Lab .*

Kujawa, Adam. 2012. You Dirty RAT! Part 1 – DarkComet. WWW document. Available at: https://blog.malwarebytes.com/threat-analysis/2012/06/you-dirty-rat-part-1-darkcomet/. [Accessed 15 February 2015]

Kursa, Miron B., and Witold R. Rudnicki. 2010. Feature selection with the Boruta package. *Journal of Statistical Software* 36.

Laaksonen, Jorma, and Erkki Oja. 1996. Classification with learning k-Nearest Neighbors. *IEEE*

Lopez, William, Humberto Guerra, Enio Pena, Erick Barrera, and Juan Sayol. 2013. Keyloggers. *Florida International University.*

Louppe, Gilles. 2014. Understanding Random Forests.

McAfee Labs . 2015. Threat Advisory CTB-Locker.

McAfee Labs. 2016. Threat Advisory Ransomware-Locky. WWW document. Available at: https://kc.mcafee.com/resources/sites/MCAFEE/content/live/PRODUCT_DOCUMENTATION/26000/PD26383/en_US/McAfee_Labs_Threat_Advisory_Ransomware-Locky.pdf. [Accessed 15 February 2017]

McAfee Labs. 2016. Threat Advisory TeslaCrypt Ransomware .

Mimoso, Michael. 2016. Master Decryption Key Released for TeslaCrypt Ransomware. WWW document. Available at: https://threatpost.com/master-decryption-key-released-for-teslacrypt-ransomware/118179/. [Accessed 15 February 2017]

Mitchell, Tom. 1997. Machine Learning. McGraw-HillScience/Engineering/Math.

Moffie, Micha, Winnie Cheng, David Kaeli, and Qin Zhao. 2006. Hunting Trojan Horses. *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*

O'Brien, Dick. 2016. Dridex: Tidal waves of spam pushing dangerous financial Trojan. *Symantec Corporation.*

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

Prasad, B. Jaya, Haritha Annangi, and Krishna Sastry Pendyala. 2016. Basic static malware analysis using open source tools.

Reddy, Krishna Sandeep, and Arun Pujari. 2006. N-gram analysis for computer virus detection.

Rieck, Konrad, Philipp Trinius, Carsten Willems, and Thorsten Holz. 2011. Automatic Analysis of Malware Behavior using Machine Learning. *Journal of Computer Security.*

Savage, Kevin, Peter Coogan, and Hon Lau. 2015. The Evolution of Ransomware. Symantec Corporation. WWW document. Available at: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the-evolution-of-ransomware.pdf. [Accessed 15 February 2017]

Schneider, Jeff. 1997. Cross Validation. *Carnegie Mellon University.*

Singhal, Priyank, and Nataasha Raul. 2015. Malware Detection Module using Machine Learning Algorithms to Assist in Centralized Security in Enterprise Networks.

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

Computer Worms: Architectures, Evasion Strategies, and Detection Mechanisms. *Journal of Information Assurance and Security* 4.

Swain, Philip H., and Hans Hauska. 1977. The Decision Tree classifier: design and potential. *IEEE Transactions on Geoscience Electronics.*

Symantec Security Response. 2016. Locky ransomware on aggressive hunt for victims. WWW document. Available at: https://www.symantec.com/connect/blogs/locky-ransomware-aggressive-hunt-victims. [Accessed 15 February 2017]

Thirumuruganathan, Saravanan. 2010. A Detailed Introduction to K-Nearest Neighbor (KNN) Algorithm. WWW document. Available at: https://saravananthirumuruganathan.wordpress.com/2010/05/17/a-detailed-introduction-to-k-nearest-neighbor-knn-algorithm/. [Accessed 15 February 2017]

Venables, W. N., and D. M. Smith. 2016. An Introduction to R.

Villeneuve, Nart, and James T. Bennett. 2014. XtremeRAT: Nuisance or Threat? WWW document. Available at: https://www.fireeye.com/blog/threat-research/2014/02/xtremerat-nuisance-or-threat.html. [Accessed 15 February 2017]

VirusTotal. 2017. VirusTotal. WWW page. Available at: https://virustotal.com/. [Accessed 15 February 2017]

APPENDICES

1. Feature Extraction Code (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(file_path):
        log.warning("%s is not a file! Skipping", file_path)
        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-
timestamp.npy
    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

            # 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

            # 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)

```
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)

```
#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.testlabels<-selected_apis[ind==2,coln]
selected_apis.trainlabels<-selected_apis[ind==1,coln]

#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")
```

4. List of MD5 hashes of malware samples

Locky

297529814d8d292594a1981fad30daa6
b97ed89e814ad91338a6bdd5f7853566
40849d82a14058cbc91d0ecea473d1d8
254c5c13dd02a9fccbe2a40ac0b04355
1d0687fe7c7c5591f1049ecb84e8cbd9
0aa56c23cff79948f977ebd1a470b4ad
85af16270f649b670cd255e6054d5388
0c0bf27bf900f3fcc6e35ba20131d344
901ef89350b60a992e1a6c67a61dcdab
fe4985beae55b054259bf14d3a3e50a7
8784d8f8eb6988bab68a1c56de740eeb
3b522c3e3fc6cf29a2c8c65a80f14a08
252957f37b8bd7a57473eab5f1a65d5c
a8344ecd79b1c1371526ee06a0a0fcd6
ea8b3acdfb0c30156977897f693f9db6
5cc3a50ee804e38a1d4546166e204544
552ba52f618ac8d1e50a5524ef15bb5c
5d2cebe73f8387466ef8534797840de0
5f0cb3259701298a76d6ca475b4e404e
9347fbbfec873aa7f19811674f80bdbe
d377f4ca5ef69de33e3c58a8d4e76803
111e581f1c3bb53c11afe0b2bd131e4a
ba7505460a9c758c25bb3de1208cf23f
bf664e4a8f36d2de1614984e6982bbb2
810c011911151d3e8a064ad44a600421
5fbdf068522177ac6392fa8f8689ed43
e81a50d312fe396641fb781a63667f3e
d35d938cccbccb5b84a19d2271c97ae7
10bdf784fd1f36aa72100aa90da22d8a
05f96e4199d83caa6f5e189016215e45
ad2022689e5de22e2e706b065e148c25
5a5817583b302c651a71f0ddfbe33a6d
9e3588c7245e6ddec33473e5b4975670
f7bc8f3b73313b238944e4812a3e4975
dce1cd7955c0352f86ecd7364f4a3fdb
5341fabc65b3984bf5e0d1718983020c
47380d71be72bb4ff55b5e51f8bdc963
296b0a81f0925c95e01839690c0934bc
d98e82be5222b3686d58a625c77ab488
f08c3b7ace25f1ce76bebd3429762ef8
264f16260d200d0501cf83220ed9a30a
5695803a695e1dc6443ea09572e6b14c
ba5a6500cfba0e674f8bd2a62a308ae5
71b8d35385ca32cd413c8f708e802c9b
e52cc2b7136c572838b8a9e2b021bd5b
17e23f44a0bcf6b27f480439714b4688
7822f2ca1be80f98649a30fe5441d0ba
5db257bb49dddf90d093d17b43eaa9c9
72dd8bc7871f07e2d6320270c60ac451
1311d9372c3550300d400c3fe83cd867
680a02bdae6724c537053f0590b731fd
89a2fcb5acc56884f13129f6143b957e
f2c79d175e2df1baf8a041802df691b7
4fb20848e9564c399d566b614caad9a4
e23f7ee88a37a90b1a3c49dc168e6fce
4febc1c6f735dd81d8225247aeaa6457
2a08b4c513ab56a3284828bfc9aedbf7
b1c957ab802f39839f2b92d7d55e7f83
5af520ad3507da22aaa756357e78eb57
124b76844281e9067654506429437545
fb469897a4536876306ae78e18409be6
a56722d826d5f222a8385cbc5666b63b
1cd414da2994719c23c85f076efed410
1b9f7d4c8a918cc8fb1cddadab9ee81b
cdd120508a1f0ff0b5b18497d67ca349
63b695765260d6d1d2a5e5fb88130dfd
3258e4be68770b76315a5059a0cb3199
1fa1023e66b05db04d22ed4f57d37651
baaaefc5706911cd0a797808e23544ae
a2c8be7f272bdd1191bdf112ba1aa9ac
3e733108c36c9d407f7a40868d251911
c9cbbb6364f51f790585aec9f5fa98cc
e4ad8906a152085e7564252b47f6c10d
d6e56a430c2c53104ca5b0cd092875b1
c6ff697b6c1b2164ede3fa5fac0e127a
b1c156ff3c59f19e30f96545bea247cf
9fd4d9c87668844d3f645b6877d64d89
885c4cfbf0b9b7956adcbd5b93688836
7e409b55d878a463e974b50c92cb172d
68aa9f8fdb7c43ebdb4a7b3a6ceb98d2
5dce19699be78fa82e32a96aee436c44
3d91e5f119093cf1639f8d38b35d1742
2e96ae4983cdec64af16788300c50e9d
28b86d53228b2f5b042db52c3a6341fe
1725b728a5225a47e3e6fc0092281071
0fb871b4b329003dd29ed674228e0206
cb3425d0e436e358a07c3f38110135a1
0c8f52995d8303837a3be33246658e0c
09f95bd2323574b6edeac8f8e349e4dd
85af825a34e5b0c000c6c4b4fa065d82
89b2bae66f6a8e24396fba2dfa062227
a6bf89594d36f2f5c499efde3c584bd0
8adbcffe2cb52628afe8d6412c1e3a06
5fc1ccd8530954f61ceeafb77e72045e
c9be9e7751b8f164d04a31a71d0199c6
a5116b31ab81b6c8fc0c9251dbb6f315
81e85dcaf482aba2f8ea047145490493
579eb4f17d08c5061d8cf71b96436a8a
2809b79768b898adc24eb276e5866ce2
1ef99356e1b53b6ccc163364a5418bf4
f5fdc2a9d330a7e607003445edd9dfc2
f0d96cecf681e1f5f1b7dbf9f6a518b6
d72d9aa76ab313d50f059774d78875de
c49177a1553b3240dfea69ee09d58b70
bf1890c2109ac0d6eb6183a98353c2ce
ac4e4c3cb5cc6c068466e937f48adcc8
90eb8948513e21a8c87f8295ac7e81f5
82f32982439cf4fa320a0f9a8e4adc98
77287dec5a92a3163c3c88ddecc8ba50
768b0a09344df69404d2466c5a45aaf0
710f6476ca3029e2017e6472b751127d
17f493da40a77f6bc7940f3166e9d89b
150ffde680083d6e8d814d93fdc5b5e8
053d6ae27d906e6303dd5604262ccd31
527290686ec5515f248d4d20c3bb29df
a2236e65f3d0849ca2b85775ade093ff
b06d9dd17c69ed2ae75d9e40b2631b42
7848d43a591033c95422f4b9eb22e071
31d2bdcd2fc117b558b54e731af02a65
8d57943a277830544fc7204a0912d937
f0f7c752758175478d275e638f3f49dc
663c6ffc9a3b9ad8168a24ca929ac3da
3280c0f144a78ef5a0dd4df35655040c
dee63e8e44bba5e6f6cb076f05d0c56b
2f6a0dd5a5967e533cd78f1942b95b50
9537e164e0eec0d55c4bd10b58efa994
4d8fedc12514af7312782e0cc66fa428
043270b914fcfc6966e20dd8cea0a60a
c75e655247b9644d512c907485b95d20
43d7a82c8317b49452cff1cc2e993dd2
66b17e85d778c8aa51ef635858faa8a3
400b1eb815c4567010ac6e908391105a
3c408d7b459151c051f8597be7e7e41c
7a3d24a705cbf7e6edad81a116c065f8
6159c5d8a54ab76dec48a795b4b73318
9d50bcbb9487430f92a500d6ed0cd8ff
598a0f26018fa23f7d46f9cd21fdc24a
3eb688eb8a4a3a87dc7c39db8ce7718f
db8e6ed8d24d2ad2f5a09b9851c25fba
c6b7f5336ae4b985b0b523f3db76adbc
513b2bad427c31cc7c6f3a225ffc15f7
44b713d31d7ddde51b0790d356cc1816
2b57d9b650820b3ca9f66ed5aadab237
f5279dbe89db9e33fa48c609f5f043c6
c57f72512fa5f47288e82054d58b9e8b
ab583c9202f705fbead50361356d660c
98cd1d2cc58142e1c662a71521229d04
86b735f30639165462f709e689daffcb
3f34b185b6e0ee6d73602305bb9d2733
3d148f33bc2e22218080d99f1f58336d
a1e5fbcfee3aa4a025954774493edab1
a01d60682ad5fadc9018908185e8cde3
9da331f4353f5b0033c162eb308a8197
1f46e31835b0f228877e161f59e6cc0a
b86d81259e15e343d4b6f64f72075d00
aff7a62522104c6dcb39463c0b1b00b6
adbc166058c872d2a61a53c86915ab78
8ed052b3c5c92727f385761786e16eb6
4812375295c39123dc03e634911adf2c
475bd8f697f7ebf88682be5458e4cdd2
3dc8d7cee33a5c2fc39c5386146a1d35
afd40dca335530ec993d9cf91be96b4c
12e15a697cdb0f1a73b05637d8b943ba
e57c0d32918eabeb319d1ee52d11df14
8a3182f0afc0d21230b9ff1a0a8d7599
8e5d13bdeb92bd7cedf828fed987882d
79386615a10ff859ee325652ef3aba0b
30896101aa3ea284d44c03289d8e34eb
0bbdd9fe374151073c54310d687431d3
01e2b6fb23d4a6b5250e95fdf47f0d01
7cf47395408c261c6b6db19a9250c230
59cc8fc8984bcda72cc4e6f9003053cd
27a6dbbc09d4fb7e0912e9fea078f5db
20b623f490914625366aab0ae7b11941
c492867c40851c748125dc5742b82801
eb49f361ed56cf58193cc3ef7bf6250a
b986b4396f001e508898baa4ba71367f
89df8a4d6fffca3f8d72eb00921b32b5
3f03b44a0981ae6b05ad7e9b32b662d4
0608285eed579359e5649881169ca920
a52ac037fcd84bac28e1243ab6442c8b
137e9311d5807974eabb5fa394de0a15
ff06ce7adf8f6cf1973a6845859ff0b5
0fe90340684fb1344c9946620ed955f3
89a2fcb5acc56884f13129f6143b957e
f2c79d175e2df1baf8a041802df691b7
4fb20848e9564c399d566b614caad9a4
e23f7ee88a37a90b1a3c49dc168e6fce
4febc1c6f735dd81d8225247aeaa6457
2a08b4c513ab56a3284828bfc9aedbf7

Teslacrypt

a30863f1a404bc2f735cc9ad862e85a9
a2fb8550db4d13c218f98862e24ef105
a1559181f4d306118f589dd86a8a3c30
2180152ae725f04088dfc9eae68066a6
a448ffac07a217b95d4c6478c487f096
9f82d05168c593b297f222e35f75d1f2
74bdc7b3c8b04d65de527ebaa5e98c6f
a4743f759b43493ff32ce5f70b45946d
dea0621fc08f5ee517579eee17ab2c14
59be3dee06e4440a0acf98a002558bd1
aaee605dc1238bd916a317ac48f4212d
a0f02c3c9bfa3845099f100931b4242b
aa6192caec3e025bb0943efa4ef28344
a28437b921610199fd8aa95c473e0d59
6834dd22796b7600deaf0b5d1927ae92
df2c0333c22768322ced8e55532dc8e8
fc2e7389ce9140a4d096858c23e854e6
f417f72aa747d2b41a6c9467a027786e
7a6029fb9d2580edbb6007b098b519ec
c1de2311ae763f305144be54402a13ed
725009eaa8a92d1d0cc46d70154939a9
2c5adf1b60ac6a0ed1c8872012ce0a82
64bfb28973ef71284140bf9a871f732d
5076b9ba5e068d07fbfd59282836ac61
4d20cc6b4021d176e806e96952ce3019
d54fca0defb36c447af3b6961711dda3
517732067df17a66f8dc2fbee8ac97c2
8949c5e942bf4c048eb8e00963bf5525
aa0ac61c36b7cff9cd85a9a6d792c45b
c6b73f8ce2f66b0a7c063c2347d732b0
6a314c7a8f70955e59aa5989587e245a
c589dca14b4333fce03b1aeb57a32b85
5ac00faf34e19005d2451c4ebf01a7e8
508d11a0f364ef375a03d3b7fbebdf11
6f58b32e2fce9e58169871880ad9fbf5
fb6ac8d602af2363ab5c5e48525838b1
e6a9e9e261b7bfa13b9f51b7cc466360
cb9cfe955cfe5fa1f0c7bb0d6624e41b
b4e7fe1420c1d22853a70813d7f3cbdf
8cd8cf63f6776b67d957d163fa042f2b
8c789cb4e1de85176d8883314e0be9f1
8737dac549e33c49facd8d000995bf97
871e464c7f9f6f8c18e5c948d8a7bd09
518e931a049d7d64c381dbebc96cf19d
2d47913035204c11128d944d78be8eb7
167142261e4d18f68ba6071d5aa4fb68
a100eff9dae36b7b3bae3ed072639d99
a97f4ba951113e2cff2aae378f96a309
08a01b8e22656b17e7effcd8ee171e5c
8758b085c23ac1dbe45c09fa2417a66a
bdd376e9659f2005e4bf48e7f98965a9
c3f42c89da95d0a85c788eccdb9b2e6a
2881ed786e725d3312121e4e328d9c32
c3c2766af1866936c43940940b94aef1
49416a772135939ff390ec5ed3382ab7
8059476333d165e44cb9526e70e0ae72
7c55b97d75b90f33f0386a904a5b2713
fa67597147f38665cf7470f9936b95cd
fd7c988278891a9e0cf174ec6c2ea182
81f9415bba1c3fa3fc732ac3ce4a94de
e562102f07c34d3ddb303ef3b2382fd6
dc0416c30fc4bd13d3527c6119cf9ace
880130a4c1a01e4c611f85576cfd9661
230c2660e50c7fd753f7a57393d1b327
21511f9f6168464fda3933de79cc381e
a4f32a5d83d34017dd806563852ce64f
38555d4660ead26e032984872c86bb12
4feb058a914f76d0de2d07f22d472f7a
abdb703ae875838616b6718966e7c463

f623aa2147c701b68f8db4e3bb36701c
c36a30b84cd95d7a88aa43e32419ee0e
b9c1e09b996165cc19150c46ca42f789
b163e8fe594af8ba6a429b32940df067
ac4d46e5c8b04bd99b454237bbf5f6fc
8c1e663762ed6f0136a6867dae3c5317
8691b47066dcb4605cf7e82b0889c3f0
7f70f850b327f27e10623d7e13058c3f
5aac7e400fefb6864971a8262a9cad06
28a2ee1c2126013c33debb4dbcace50a
e8c51d8d429aeee10fd983a2fc279242
dee1f217663389710c3b729386d68339
4d8339c4ac3225102e60e4885b443268
a601cbb0a15bf9cb4cea80be4b6dcfe8
a5bcb278884567194108fedcb797e7dc
cf1641bd5394298e6cadf07812fa700c
267f81735a124928840d7482daf5010e
f27f3bd810aece963a520583c6481a88
c7eef7fb8f346f150cc23db56eb6ebce
061b069a8207931d64b2be0f666fc5c4
96501faa103e99a3408494bb95b04f78
5b08230bd076c020acd55871d0a50c79
1dc7fb65a936731dbcaed723419a12f8
3c524fa4a569ca559302e289918cfccd
18b1414e846fa91fa7246c80a4d20f21
0c93686d4c166f2beec86b507f6433c9
79f170cd385b217275e7c7db63899eae
c71be56223ce2b7ebf5a10bc10750793
c385dcd7ec1f80efcc31c8d6bd0e8364
9b3b9a40b86c7864819fe6b54e0f1725
7a0e726d058c1d95c8e09b4394d4e5ea
795352e779a0105e4c644df3085848a5
78d8fadc8ddd5f17dcac4411f145c92b
23e355d8d268c4795e95500d22515344
1914acc9110eb0f72c000bc1a1061497
50aab7f0c52691e1e66d544e43e16987
ca03464f0ed7235e14485c8e753ea572
48471c25da611c4a50ede7e7408240f4
9659d8de6bd1a3648c7a67ee538683e4
60f4fbc0a58f146e8fd6d90b8d57c271
2696bfb20ad7cff3fe98cdd7219b88a5
118064f032f310d3f4af292600f7c7b6
fc00d65b2fc6d46826a4deaebac54896
e2cf4230402ab26407a344697e67c243
a95ffcb339e7daa98a2f68b65eeb64f1
e98785089105baacf3d655edd41bac61
daac7d7a60bdc62d7b18f6cc9658e6e0
b1021e2e00dc2dc2745b8c6f3a9abc60
5d8dab0445d7b7db7bdb0f335b097949
54a651b3cc5255e0b42b7c3c50c65cd9
20d8043b4b4d80de696c009afe7fb671
fa2e23aaf5ab9069ee8b3fad0c3aa591
f88c12f2ccd31b7877031d8a22d652ed
ec9a8118e2473027cec05ebaadd074f5
c3cb934d93db52b22c2ce7d03abe8417
8f6a85eb0c58837a48a7f9163e5d30d1
055e612b2818622f50967cf098427c17
923e4997bc57d1ad633cfbc029e2ecdd
d5975f208ac5783f4b58e8632cb17c47
f50f693ba762bfb674004a49560a23af
da9e982175fc1a331e644f4e4fdba26c
d907a9616fc14126653d03772fc7de7a
d5289115d669750f4f8f448842eee8a0
bf7118e2fe12f38fa2c41ad3cc843344
bb75e0ae806040be55b40d452686d770
57daafc94241d097f7b7462fdb46522b
1c910936748d1c317ada333a06445330
0ea102aa6196a11eb84cb5a4ecb60f33
f1c1bfa9c4544b05d642fe2a865abc36
a1f5f6c29895a446b6eb831075cfaec8
aa7092e36d6ac9f885b8caaf70b57ed4
224ffdd75e2db700e0816bd27cb128d1
12562a42cd391b27ddafcf8a0b0cfde3
828af987a6cc99586a72a017bb0ee799
0f3f0d90419cc8c91ddcb5c430867707
3ad4b974967a3da9767ca80e3227ae4d
640a96f41e1691254e33372a5330ce94
5f1db9e4cb450d750124e244b520c514
d6609f0e82f65102b7d33417f2e8c599
778d9b3bafbd4a0f74cc32f2483fe0f5
52c3ee0d5bf15949dbb082531f865e9e
b0dfa978aaba1f0ae0c79d05b8f804e1
04001a95c39716be3cca725fed1c7144
e3241e5075c58aeba64d8c9496f53b4d
7444ace98a3e2e588a39fe2d54180530
fc1698a45c58bb95f9e47db83ad4a9f9
1cedac81bec37becff994f3bd16b1a92
ed07cf9bb0776107b2c3eedd2fdb8064
08362cb614a2077433f18a43eee06820
7b1b891dc2e23fd6207292f6b0c63ed6
aaf3ac53fb8961ffc70d29bdbf1d718d
96469329287b7e2cd7fdfd916addc1a2
a5a704040ad779d2e70db4780a9ed219
e058ecc0e3853de1f8b727f3cf34dce4
bc8be30f7c39e1cf5ceba3670cd8da25
d5f684b45126d5303c692a516515e845
48ffaf220f167c19ad9888dba639c0d9
77d66df823582d46b5b41e6837b89ac3
53339edca63641c414ec1eecc6a662ba
56dd14e03a486ee2c385318a34416a35
4742ed29ab69e60405a75178c6685c9d
15fac9dd13c580ef22a5ca980bf1f416
5c4b120733b7dbba4042ee4df4a53bcb
25c1d151e7377c0cac6572edd3df4a90
06fbbe54ff699d90197e4bcd730571d2
b9ad0e1c2e7b129645ee202ba833c296
1571350e00ba56b2bf23ac751053639b
b9f41073806c83a9d44c0c66ce8b55eb
a8aea9e1d840359af9d7ce6155ff244b
d79d46621fe7b9fbbf1bbefba9c52a5f
8093d1dd604dd0890a18d50fceccbb4c
7af90c3b2578dd9b15078c3ba9ddf7c5
ad60fe328fc3b24f6cc7a5bdb34e0ee3
1938b6e247a686839564ddf99a1c27f8
0fce6304e09e798303677e49d12c2322
715bf2f0f539b49f5eabcd784ac4175c
1862ac546f78c27263300645449d6d8d
7af9aacde8bdd650188817fc4ebe1599
c4b14d1478cb8f2f03be946880c8eb96
59e7926695207e1097ec45a37ea2aafb
156aca02760e0d7ed1934b666794c45b
776e822a7d178b76143ce058c87fede6
951c86cd98b1e181503167299d410857
7df5f6d611eb67f481a95804c5ce0630
c731bebaf093da2c82d893df2b58c8fc
32f6caafc8aec1b34f243d029cc1e885
0cdfe31534220e6c69bf47943aeaa712
d58e0347714ba9dfa46bc33691a02d87
8e0b2797a7f9d79e39a73ad54a73b0d9
db9a3ebf29d902e3177a26aa8dbc2af2
ea7a382a75db24dad6014a4e9c5e0ceb
233be9c321ebcdbadf27fc61167d45ae
36e54247977b339983349daf471ad8cd
2a511414a8e5a332c4fe349a47d20ba4
830d9f6ca8d9494c749fbec24d3269eb
6981409964483b4b7624c1a85f575e21
41a740d9622767940fb511a6b3bf1adf
2b9f62978f6e7b72b07b8f1af0c85730
a05473b31a0446db76c512177d21c474
8cba1524bea06be9c5f093a0031c32f8
aec45f2ca076216d0768ea518b9b6c96
3d95231b71f286af199fb4fea138a29c
cb94d44b89256b00a9fd33320ca3d4d7
5d5b6c2ef3a7b934ed657681f09abe5e
8c0a09ae8a711bb0a2dea9fb2f32178a
6e72482c91f40143f2014c3fc4bfca0c
b0b6648542bd8557260e16299baaf72a
d15eebbfd21479858c044faca86b7756
555ea3963b253263ad89ffcc155d67a3
6f9532c7b7d3e508732ded880e6cf831
dab5ecba74d7113e73a9a625d404edec
fbfd0c1102cb745e583986f3e9cf8ac8
729aedbd826f86b213cfcde2fc1e1743
89b14b6eccf58083abf4557dc42ecc08
c7241928868b6cabb6a35c90f274bf8a
285f9f45e19b7933b39d23b9a3ed0563
4d93fa2270b31acea407a786ce412c21
c3a4b8692c6bdb0bce16067f150419da
071313017f6e276ba2a800ccf0362014
ff77c722471c1db1084f26a2dfaf64fd
f50c3962458aa054a20d65b061f17bff
f21bba7e1dd8425cfab2de19e181bd9b
f19d4ca93aee0422e49cd3158ed6b9cb
e282a97cfb1ed590c549c5d659390e59
da28166f9ff6bbea1c23b63dcba842f1
d7a72a833bbca58196537a6345b9f4ed
d6551e0503967dc7a5a02dad6b1d35f6
b96fd404534f155f7566947308fd220a
be224f27fc90b7f889268eecb54a72a0
b6cc780f0fbf531f19ce93e29c3148bd
b66c5bd925434d5c9777fbf4428e59d1
b5fb5edf9105c9c3d16b1010f6ab5fb0
b1602e09029ce232bf41535919b6964e
b0c2851fc7e9010469b532bab91b15c4
abcab0cd2457117828a564c6135c0b91
a56ad8c98d10a1dc1287983bcc6e11fa
a21f0dede5ef472164284a1522597e04
a0e31da4616857c6e8746d5505b8e43c
948b2d66e8db1fb7f4c017ce4e381098
9317cd510d54508682f08b0ff0b5d7c7
89dba5be2371073843a4cb8fba81184a
893df996cf724f57f9fa153eb8cf3c43
716f15fa7b39ad8d3c46e61096abc7cb
6bdbfa1cde59a0189116ca91760d9328
5ff6f37db3d56c3abe8dbfb938b11262
55e2a9d59a3dc13bad9646a8314f1d57
55b7a77b7c43cfa416a679d8bfa6dfbb
526cdcfd23336e5e7254422b4bf9ea83
4f8001b62e13e04c243b76ddb095eb1d
48cc26a4fcc6a3a09bd81bf8bf773531
43ae902f95cd48f77a744758b8e0b32e
400cc969285639696de9de4e335c7d90
3d89a9ffa5df043f1a1571011952378d
35a37f38f99ea94ee65fd0428cdabbcc
355441d76229368f22e0d903558f1c24
2e1ed3870d996dd89c1d1db3409424ad
2a481a0c5d2bfe1e7b7c88f1f525b0d8
1e1ef2c69e799b5feffa5d797efb7773
1b86d6ad044616452ebf604d6a12b280
1b46211a12fc2e419bc77feaddc68053
192ef3a468cad8d147b87bbe7520bb87
15be4aea16ca93fd530d60e36b795965
15333f8519cdab0414a6fd49e3dcf2b3
0ed44fc3abf67d9dffb4be4592e92e5f
0cf29aa7447ef1dce0ed36600f13d7fe
09d52b3aaaa1dc5d88a3172c1d540685
073053cdb253dd1057f4a2a442a3ee95
005a6963097536bb687192aa0247ef3a
e40299365df054a684c642dcde2cd2c8
d38cc7eae608e61e3e117babfaa72e3f
a092c96335a0008577f9165cde69d1db
e8d5e913475643180b2f5d4ce6522b0c
9031ce1d24ea20673d49832ae76f967d
f22601593da29f220fac234f3031087a
837bfbcae22ce95f2932e572e44b099e
3fa03d0ffd0c1e44cb3139fd3f722faa
bc2ed64cb7d0243c7b3c131263077293
7bef5cfb592e6cfa35fbeae5acaa15d0
7eeea350aba999189fdefaa8666cfee7
bcdda69fb415df35b44818eb788b918d
efc68fc517db1d700434d3d7cafb592c
84a2fdc47c1aa5853c139aea37ad0016
6b7ba8ab030d6b4a4700e8c7b9b3a40b
99333789293e2198b5c75f69cf978e8f
8952377a0513c24cd2a00e5946d45103
cab329ec48c90638556a99e1654d0509
794d2e712252bc8bc682f25cc66c3f3c
1384c22e8053e329cd8b1cf2ef513430
fe84dd31332a4efcd474011be27ed09b
baa493c7a1ebd3b10947511ab46bf521
c44ea1e978b5ce9fd098735395273749
5d3ee1052b86c6386235e561972d4ad7
dc9ca33654a83f584b5baf8ba48e51ab
ff90f59d7fd1e747197353c5ac440d45
73f78062c4ecee89ecbc8f23f590e50e

Vawtrak

43e2354bb9bab6614ea1f8b154a564f4
9c6a5663d83c38781d4bc0eac2c9890a
4826e1b51599e3eeaa792e9621170324
e930b8184305aa81965cac7f1975851f
34976648b44273c0b336d2ef89e672db
25f813e97409bf7808756f1913b11102
142ad2c753f1929a3407952fc8ac147d
262a1a847173d3b151a9e049f06f948e
786c43da9212a35dcd3364d9a09fe1b3
34080a4e7c2bb069e525e22e555f60dd
209480561bbd613503a0950211588f4b
1dd40a9223e0c43e4e5890aa84da1844
d0583d0c51aae77de743cfb29ecac4f9
7ccc57f92ee1132e30141f22bbb385db
ccc611f636bb96bb9e34c3da97f3b8f8
99f8252f6f396993fde32c1dbbbae61a
31b1e70519d0a8ac1303ea89a0d817dc
8927105aaf53fbb0495be81835474d74
16b691b9c41227fb9aaf592d7f49c722
7d9b38fad4992247cab2663a1e6ed137
6359dffafaf53f1d4b7e2d548a9556cf
28b577ce059b5c3851b469911ca637aa
1d59afe9be5899c51c4f62aaa6536e5c
608c9297e685d1037ea72b55474cd9cf
d7b0bfb55d8d46d8f661a4cb46c531ae
845ced65ee8d3ada63fb940f4dfd4e51

199d642f5c50780045085cb5992a52fd
f0bdd217d0552bfd73e6d217cb65a739
3c8f1e08e774dd503d7528a1d6d49951
6edaef466a97955a842f54e53f205991
86af5b1b003fa2a570dc45ca247a6274
6b87d33b169986cb34f913c14a547f75
110458278211d7f6f29180a78fa125c7
4172363f9ef187f0b26e04f9e331501e
842e2cbdea3abc786332e1eeff20a59a
973fb4955add4ca88d4b661dfdaf6edc
c202883ebe5033041aeb9dab8c635f1a
048d559df99a7fee82fe5fd4dfee900a
bf64cbdbcdfff2d00d75c620cc6320ec
8ee24f9715b6cc5711557d59a1f10581

Zeus

e7c054ea8bc2f66e914ef82841d329fc
86255ec982e822f6b57855d3866618ae
50220851ac85a9422c35966b433c203b
8fad5dfac3671d90dd12792ec81fe595
237031fc5cf6b5ed59d8e750b860341f
7e3b8c6062f7f11fef7cd66d068539c7
11a1c3b8be7a08c1b547a5500efe17d5
70a0f9cef4d7a4952eb659b049e98fc7
82f34481e82f289f89ef69e4eb2abb3a
14a18b30c40f5a4fafe08e0c21cc5844
353f3b54de9ecfd82c63a2aeaf1c3b9c
2c0244c28036f9cb5f9a703c8b329f2f
461f8cb7c8f1dd63b062fe726ea764e2
96383909da192e14760de588761e38ac
46ab2d15b560b7a07d39862907290220
d725561817d04a1dd0c889781613b577
bd6466701c9e93ab24d77c34d44106a7
09eb0efbb48e7efe2e19e71edd655f3e
199d04e319f6f8c1e88ab3ddc7ed28d5
392b86e7d4c9f28e98862e39cff6e49b
a25263c96b548b76031d96b43fe46b08
1e0058d2e69f3bc4b961451710e2fa06
65b16d40f024b5c1cf8676dc1c252d56
ba57db487bb18b15217cbb08c923da50
338caa225b3906be3bf5399d8cb74df9
dcdf3aaed2047d2bc746ab3200667261
c30825c55ddd1b3d93ee6141d44c78ef
d2d969b5db07a1dcffab0831907d31b5
322c39b56988f5a35e64c54633c196ee
3a2c75fef3be79b5d8662f121f85b4bb
3356ebab1bc8aa9e6212f584bdfc7566
78c00cd8930229a1d34b334f983a633b
6da10c3719c7bfc5617f6095d08854cc
122d32cf91a5f6a545496e0c7c64355f
63992249e966ff33d7555e887ce28595
5054c0c2dad7207eb1aa69f5c48c978f
b73aa307e8c2328f6a7dfde1a1f024fc
065e6b516c4fab893826103db6aeb5dc
be39759c2e6f2685097deae282692851
8802d13595da8294c84821e5e3086442
e89261b86d55db32261ea9117fefb1aa
9b2162f6148f7ba9a15e2b2424952973
d87a03973f8cb42b90a573f831d29bd3
493b3700a1ac3b5b872bf2a516bcb701
0db7da4e3489ba8a2ddfb128422daee2
b2b1325cbfcc52be62e38fb99713b1f3
d86ec2bf5962d1254e08458b17ff9594
9f890fe67151372e2dcb34d4329eacc6
eccfe46a97deef63172aac0ae8771d9a
805df9572b345cc8691198ed1caba924
ac2d2fe0bd0c40db038c88cca4a3296f
33fa98383d855527c1a166ac61f92ec2
ebffa1e446ac21950941ae3463aa2df2
1fa4764c0c1eae57af50d4a5277886aa
4e9a5f7e45043f7ce0e2822b947fc117
2cb6faf81fc9cb701e71d0497a25f1d1
177e77d48bdf6424eaf0bbbff2905236
29c0e993e5ff6106d93be0b54282831f
7b2c587c79dfbdb60d71c0144a3e2ece
7331895ea0d778ffef3ab95d3e1c355b
6c3fd4e592eb5e0f5d8b4a2f76d9fa8e
e94de39e9a23550f4a8a18e6cf77323b

CTB-Locker

7c9c4037d574cfea519c4c3d43d733bb
67cf92d2387b864723bcb4dc5178eb15
d5259f68bf8750ad7426b423f028e700
ccb02c979345ca34fba61c38465d5ea0
2bbaa69d4ee1c512eaeee74bf21ea234
12673429ef910fc9957248919784cd89
c4b29d3345ed0b7a0b284764f3446396
aadf8bb8770d563aa27eadc57ef238e9
db0a2be5b0eb4603fada6e6f79f3d267
3be529d7c05913ef0571a6f4fbf7d946
555eaa028a55704f3200614a01d0a464
b4a012e465f845bc7ac8b441bec2d2c3
f87208a911d9d1a3793914a649dac97e
78b22232b87e0598e8f6f584f1883c87
6d0509c64d650c95b0e132567c063eb2
497b225179a648c3fcec8773351991b8
1011a4fa57954818d4e378a2af9fc3d6
77fac4194a04d2bbd9b4503044f4250c
fa51284e30ae0547fec613aecf61191a
5bf3e5258ec9efe29f92acbac924c451
cf98e53ef516ef190ff0039073a048dd
b6cdfdef61c8acbcb72b43547eb7aab4
a4c64aefb73d3623521d91d5d63cdb45
01a466176732f7050a2853334f2752b8
44351f134a282f98359766ca2cd03e14
428285ac08507dcaefe93d269247506d
73c570a78ca50abea3a623c84fa69daa
33e3869b8f83e4f6e7cf57e37e243bd7
442d975dbb126cbc2e67e87e3fbdb3e7
18770c7b60ea583d6a7f24f87ac1ba79
1bf842a82acfa384cff8cafcc70847fe
2d1626395dd249c3f0945597c3f6d82a
3f40a4f7e25909f83fb3c9eea43fc5a6
afd71522bec434cdc6b88846a9492aac
7c5880d358198f2b987335759dde6ac7
d769cb11f4779d0054fd9a877abab214
9df802d586aa99385940f62f4bf4b1ee
54fde8486cc14f738f71cc14d1e696ac
bcfd0793e4f8839cb1cfbc9baafdce6a
2004713813fc19550489da11070ed5de
ff7d5a564a87d6802b84c4181aa1b374
a2683c2bd2908a866823ebbfbfa53739
cc408fae00a5cc333e60df1d80d76785
3eee2f067500bd40fd6ad7e073ef0395
8b19b6588b96f8ff0a64dc9beb531fd7
d286b84a6d50b4b3e2c21e21ce8f8427
7aec61bbb9178868ed8b98f4f924ed84
0bcebedb7d37e6faca9ebe0234cda920
c0efb6d55a141faf7d9de077bdff992c
5859f85c79b316926de3e5e4c719d958
a7474add15ec31f416735bdde6f008ee
3f1b5f5d4e2970c6906b4c7dd83d2a26
188b5f83ca6ea33bf99af52e85203f42
495429bcd88402d8fe7e9213a17f1d00
a5b9efd7aae56ab5b3e166b54400cea8
d8a6831e73bedcd7f09696e2dfe3f732
957fd7a1c3aaffff456feb8c157d2517
a65a63148e6f7b11eddb8f54a7fb58a5
a0ad6de0377bbd43cc993d1f08165653
7c9c4037d574cfea519c4c3d43d733bb
cc2d120c65a79bf49a3e7bdbee122e19
f1a88bfcf2db8fe01f2805e4b4067152
67cf92d2387b864723bcb4dc5178eb15
90646b00de8f964e6454d20d06315567
c4b29d3345ed0b7a0b284764f3446396
aadf8bb8770d563aa27eadc57ef238e9
db0a2be5b0eb4603fada6e6f79f3d267
3be529d7c05913ef0571a6f4fbf7d946
555eaa028a55704f3200614a01d0a464
0ddfb2e0506d73169ec9e75b80941133
b4a012e465f845bc7ac8b441bec2d2c3
f87208a911d9d1a3793914a649dac97e
78b22232b87e0598e8f6f584f1883c87
6d0509c64d650c95b0e132567c063eb2
497b225179a648c3fcec8773351991b8
15321c41f6cd553e1c533444a380338e
1011a4fa57954818d4e378a2af9fc3d6
77fac4194a04d2bbd9b4503044f4250c
fa51284e30ae0547fec613aecf61191a
5bf3e5258ec9efe29f92acbac924c451
8c2a701dc7b1cb16f77ff64600538568
0008321a7143bd05182043aba53e2d8d
a4c64aefb73d3623521d91d5d63cdb45
a60b271c13baa2651735f6e3fc9ae4a9
428285ac08507dcaefe93d269247506d
13768dd756e873f3d0d613940fdebddc
33e3869b8f83e4f6e7cf57e37e243bd7
442d975dbb126cbc2e67e87e3fbdb3e7
1bf842a82acfa384cff8cafcc70847fe
2d1626395dd249c3f0945597c3f6d82a
5afcd5cc66ab0ddf68b1368d356bcea3
428f97739da47241b8c88b527ba4dceb
3f40a4f7e25909f83fb3c9eea43fc5a6
afd71522bec434cdc6b88846a9492aac
7c5880d358198f2b987335759dde6ac7
d769cb11f4779d0054fd9a877abab214
a488ab820757bd9933b4c12bf620f03e
34feacac7f909d5f5433eea8d88fe6ad
9df802d586aa99385940f62f4bf4b1ee
54fde8486cc14f738f71cc14d1e696ac
bcfd0793e4f8839cb1cfbc9baafdce6a
2004713813fc19550489da11070ed5de
0aef2e849be6782b64aa95987c92e450
a2683c2bd2908a866823ebbfbfa53739
cc408fae00a5cc333e60df1d80d76785
3eee2f067500bd40fd6ad7e073ef0395
8b19b6588b96f8ff0a64dc9beb531fd7
d286b84a6d50b4b3e2c21e21ce8f8427
7aec61bbb9178868ed8b98f4f924ed84
afff0b9b22c7b9ef8cc10ba5b3a16101
0bcebedb7d37e6faca9ebe0234cda920
c0efb6d55a141faf7d9de077bdff992c
40ce00566109565d499b53a150716303
5859f85c79b316926de3e5e4c719d958
a7474add15ec31f416735bdde6f008ee
3f1b5f5d4e2970c6906b4c7dd83d2a26
188b5f83ca6ea33bf99af52e85203f42
9419edbf565f67d46e087ac343447607
b43b19699691f198be2128936f6bc872
495429bcd88402d8fe7e9213a17f1d00
dabfbf466bb46fa6c07618c08f1dd97f
812b538c94aa1703b2b0d457ffc11a46
d8a6831e73bedcd7f09696e2dfe3f732
88b1309efa06d2282d151aee4fb534de
957fd7a1c3aaffff456feb8c157d2517
50409d8c532f6b9ad43deb90a6a7f5d5
06cc11966ab4d0154faaf323fa08df89
f73cf0a85e51e5cf95fd1d4162727795
e47ff86e1a3daef8d01a3410dfc84b2b

CyberGate

e6b37fe5775812b5006cb6a2bb89977c
31fab5d4dc9e7baba161e6eca79af442
50198b5b3b60bed98a6531fbed2252a0
532265c36a89daeb95a025395a817348
d3cce60e0f2b7d92ccbfc9dc1bd7ad0f
60a6949a16edb92a8501132939008127
8e82122767022b7d49f1f2d43f7040dd
9869f4ef5a911595156ca61991126212
9a15502f62ff537618841b4d4baafcf0
cbfae9234b8dd18a3d5cf806e3b46263
7ec7d9c3927d668c81f0d41c579a0df4
e2805cfff9fffd24e6abdfce3643afad
7d1788f05546128ca230d4a925472907
fa1a792d0f253c4c0b105122cfc0b46a
6cd39faad8a780ce2d66cefdec058899
dbb099b05177933212f8262b936d1cbf
feb6bcdad25e61badf8146ef6d85d9ea
a15bbf7546400af475bc0f460c866ddd
e3439afd875ccfc90b741379dcdad65f
46604a70b623984862fb89af33f8ad3d
e58ea7a6dd5c56629608d6413ef3501b
63041fa934971c3829aa68b7ca29473e
0ba36b98bcfb6d98872fca6e2eb91e6f
3e4e1b731cd46e6a854bdcf268ea3161
80a9bc2361bbd24b21c18df0dc4f2580
9706e40a6df6940485234ac4c6991eeb
65b1ebf391737d64438dbe9279bcd096
3be00ae45f84baf74ada5ed7e87cf6fd
b162a2ace62e9c444530766a06594f21
6b805a28b0670828062416df99faea2a
6f09cafb88a4b7a21711e7fa877c3d03
52978198e3d3d5f0a614f7c0fc236e8c
a6798cfb887d15300a769f8cabe13cf0
5812f73c70df432fd61709b101be1cd7
a90848af417cd6cd1342fcdb3a2d23b4
6fd33787c1344bdedc931434599cc960
a54e58e3239d90bfcdc4b18f69cc6c67
ddeca23b697762faa189f94a6fc158b2
98796034b56125cdf61cdaf2c9bb8f10
1c60124c9a8e604078f519c4dfea855c
1a7f51bbf414dec32b6958f6f291184a
a33c8fa06bcc84393d0d7bcefddc4d70
2c8bdb2620e0cb5844a32f4a2e134f04
6d2fb50599068772d6c3ac508bfb9b41
044237a829c8298d996dd0c302be1853
e41754c51c658e1e56e41bfe656c0781
8803d50c9065603ee66a17e25c1212b0
ddca2e3af7db65056f76a5f2178620f7
9366855dc179cd99e94c18696e9625d6
26dc848575bc3204627dea150b10a2e8

f2a4feb0ce6aeeb9b981013b51756fde
eebc4560fe3677a4ee4c9b040c6647ac
0123c976998077f97c647c6effd71dd1
3525cfb74a1d5c432333623f81e81cc1
49c4bc1473cc9069d79d1a3c0163ef72
e20cbd5763fcfd9da3f96dc940c96b08
c6b7b80f2ecf55b69b0dcf6ba77a0040
f05f3a44e68dbd70644814d3ed2d29bd
85861861d22459de82cdd43951819388
f4d5c811c84340774ff477c90352d860
c00e759bcd9f2f094751cf3adfb18b9e
997c382ab3baca83179620010e6a66b7
93c91852b419384d8f287c549c937a93
72ff8ea46c059831d8bea31e11310f10
5001f11413423fead58ea4942c0d4977
7bc369eb8c71bb7b19a7c6ea07a51235
9c8ae93bb78d89ecc8b88b706caa5f0c
c60d2508c3bc53450a7a3f8a5b0d62c6
3f357aaea658adfcfd01d9939171738e
68b7a8da0f88c2a990dd921978bbe7cc
6636395c024a60243d057c3cabc676e0
430b161a360a2580561941bb867e55ec
94a42983d9dc0900c18e8428b3157f71
3e7734f3b833db26f0a46765726c604c
35a162fd3f39f8e7369fe60580352da4
3a6438b5da1cf8e8991e5acf558a8fcb
f5708c09e07692493f8a7fdfbc524c17
c346fcdedf2dbde83bea4b3f04a70a5d
54ee9bbd9f5b382e1871eeb88d67abd1
61df91d6efac0cf943c75edfa776c511
0c83af8102f81c1d4dcb2a8e3fca45cd
24d3daab0ea6408ad7a04a362f17f936
99dbdefe6c69233dc101932d62aea219
79d089d8c31207fbd85b3c94763f43b8
0cd0d9802e21c72f20d6c54afaaa848c
2a281e4c52a17677692d6aa7258832de
252e1f94363fc1cae2e5f3f9d5c93139
ab261bd57fd7da354025fcbb88c49313
815fbe1ac2407a16b058bb44a771d024
562049dfc5e2c2e9b90cfc01bd5875c1
f0e10b91a56e761b5c88ab14f0798143
97cb703cfcdd52d3376ee26b45429e0e
a5ba70c3308684c0f3cd69f1bd2ac170
268b0bdb8858d15ec59900922f748de2
66975096369486b3ee667fb5079dc3b9
6a78182cfd46b02af525270a8daa1d31
8c968d7fb508aab8d950ee644e734464
a049d274d5a2debf4433e3035abb2ac7
92b00b54ce1bda07297d7b9b0212bb3b
2b82b9b7b0d5ab95d2e5455e6883b3e0
91c1a2a23a3d27eed35331e905772373
fde8b828e5119808a24943a28a2a0970
2b24f22b6a5c8b82d26adf4dcf23db98
36b671655d69aaec27b35a988a589b01
38389037778ccc1692eca9af3b608273
5b1e7f665f959d79d9a05193852652e1
2c9bc4b93bdbe9c4320680e825c32593
497b6e3e2d62a2606dec78ea3d2a1521
cb4e1ce2bbe7e2ccf751d195836895de
15738eae47ab9f00cd8a76c2bdcfc7ce
8c282f0dac553cb2883715640d2a4746
9640691cb9f549b70912aa3be320ba50
626c62e7ef237570c54af289a4fe9e70
831a91bf2f0169f5d489dfa9bfd11ce1
97d61faa67393fbd149443a170973b90
08cb0d4fe71cc770ff2a84f8946e21c9
e27f671aeacaf4306a2b21ebba57de6b
a65846face57e70e2677d35fdd9ad650
e55541e4dcc41d6efaed01dda3a72ee1
a15cd1eb9417d3bb9191bdf76047b5a5
d3da1bbbf3763db6d0d930ea7994cb56
a092c6c641bc63b0c53e42ecc79f331a
969c853f98a8e68557528ea3205f07db
ec337f19a29dd611753ed5e9360ab272
72746ee99eb41078cce9c62010a0ef3d
b77b8bed0d72e4c130b6f1160c3092c3
04a6c4810effc7873e7271fb64a8b02b
c33284f281d9ccc63386ce5ee78f542c
ecdece3b03435e83f2c07589bd15c140
98b411c471e1457ca05d596cccc69360
0b9a21eebf9e3a0371de196a5d98324a
91ea948fda1672570d822a7b7c24a11b
a161dd319e15d52ce489106d95635150
9a1d8eb265d0ce2f3f17416c5dd1abc0
d510a80846a68408615844dc0dd8f168
4fe25d785ebb9c7a540da87782b16a70
9384f08eb288b6614814224348252526
9aedb9c8e3f8f5d8a6ab74b7b0eaa589
412bb3c0871d6cb2bce6dbb7a07bfc71
bf8b08067f6db63df667023d07a290b1
889d2e59aae86a5b733fb2d741582a8d
7abfc10ec74d263995592fd60899a25f
0817f8f0015e02ec5bfe897de16cbe21
6d637cc5b02538a7cb49a9d15c80fa4f
61b9dcc35994b303eb553f7601788e16
756192b93ee26ce2f875102b01d89380
e31fcea6344662d97cd8769515f06c60
761148162c3976ad61b6ffb0b4954af9
75e64db9af83faeeb8457514f78fa460
871a2a879d5548105f3cf08f3a675168
19586bb2b7ed5e35155340623c3b0088
f8f07d949eff7b72b7cb84fc52b6dfc7
b4da8bee85a875343c9c2ca4a52b362d
21a66213adf0ba37c737acd455db98d1
813ba07affe9c2ea74cf8ae388260884
d4407496716fbdf5186703675d0a1cd1
4e6e214c6c23ca2d741dcab87464c527
a620a37f802fe4ec3b620531099a185a
04667561eaabaa9d2068568b9e4001bd
55c89fbd4225b37278cfd8293b952180
9677016487cdf4c1f037d833a2b7d120
b03f30ca09096f87eddc2d19b9c7ff24
1bc1d648625e6ace3e1f4cb55cd4a180
85678cc81a645fdd52c1f51a8f5a38f7
e68e02368da1806c1f81b6325c7cd7e0
28c1f66bfe88f02db40c36b0b30d0ba1
9492fbe49368d26658a3ba2bd596a8ed
988f7afca81eb153d3945c734f5aeb21
eb498c5dea684e9a9ed7da5edd12e472
e5df9034445de568960b74d51962ef6d
3bb019d29bb949a6f9d2bbe3eaae3c3a
801c05a5d61883df78d04f2fb5612d5d

Dridex

9e06e2fcda95624e24207112632934de
5b4c1b0e5c0c375594018a79b2a7a4be
a78a2cfcb76422b16936537bc5296996
b382486264996132983c279b056bab2a
dfa8814b043cbd03bbd56fe6783a159d
075d6ac8bd9c1d1e0ba6aca7ca98bfab
6a46f5c6d1417f159a0134941da35cca
0111b2bd0543d5e35acde388ed8b62eb
debe7758f06568d1a1d98f26ad8ef9db
017d9d43baf8dc2d8451d0ef2e9a07f6
2e0c328aae6abfb19bf02e0fbc5dea93
2ca3d7c9f95cbb42efa447d1cbe9a885
3b1a28c2064bacc292e473c9cd49390e
8f3063ef8032799f71507b8f88f8a1c5
0ed7cdd5df7bf46fac39ae32478203bb
59765f9d900ca3788287b683e39e01ef
1f259a88f61e45cc6f357f2fc8dacb9c
2845499946fd5882f94cc9a4375b364a
cbb76581026efc205559598918841b32
3a6ba8f950ce66247f8ecad7768eddaf
d92147110ed2fcc0de8a4fa8ce14a558
d57f52a02404302ce2f27b7f693ca001
f6df80f3671d48a0af931d3b8fd290d3
e6cabc9b892f5b8362cbc8617228161b
d938a67f1cab993ce0afe47c699fa907
d4c3e289e5c2240b4bc06e344be6e5b6
b227c91fbc1ba56e9f01ab4f1e2e502f
98b7f9ac486c577b8c3f517a03faf31e
db158a28d5f831cc1450ce1f69e94459
265f7a370f09fe912dfcb465ad79f2a4
882435fc2acf984e252cf595e244acc6
7350d128c6a5606bc3ec43ca3cab7f27
d9501e0adfdf8539c5e0cf4bef62ee2c
f4b0f7ba911d067e754a3028b6174aee
ba72f454bc216c61313c8eedf55f1082
db65c736eff94e7ee5924927607f0be4
dec85f017d2905f82497f31b3b551222
da05fe0a977a9c30d232719288944a9a
de6a14ecec8ae8150048f90d1eca363d
d63a59bc621e0380eaec5bd07e47c4a9
faf8433ef3b9c6fa53f13129b8d79ef3
1a4543034cc6669184ecd4682b624198
49919b3662efc214bb9a9d1800303fc0
ddfab0500bcdc5c098ad7fd2a1ce82ec
a37072400f9efde4dd0ff39d3a5ab15c
b6418505dea2ddfd514cb4d0201bb955
6d934dee9ef917d58d3021f5fda66e91
dd4d75cde2d7ca60ee16ed717d0dbdb0
94db4504de4e3a7c00ce3deeada87a33
590f83135ecfcca3f0d1d741619133f3
359a0b19f195dd9881dc8b40465b2970
0c8a9a1ff7c1873a44a633bff1340bf8
cef009cd8b1d72f3467dcaca691c82b8
c6fe297e17693171457476f52c3eeaef
beeaa3525de08777943964047c026fc8
abc7944ddc2e28b154ab4c6f18fbcff9
dd69bdeffbb8dc89404d21ed6a538b3e
c084fe0271d68f63f738744c38f12251
f49a7d462bd90214712153f97485a627
81c1830e41b2da2e62ab0437c94d6fae
b734f059295c6c41dc5965d1f4357bbf
2ecf67958ed874401752344c3d0dbd74
5437946a684c7720397df91ef7a2a2f3
1827fbcddcafba6ce75eb5b8652332b6
031fb0ab676cf7b8dcb460ccdf0dd8d6
2934c524678e7e1447653e72a1e8ca3b
b36c03005b7a8a825ac0e2319a87d8bf
32150a6d2e9857aa311cbadf5ac8fd50
2510bbe7510051c28caccd58ddb52007
005f267003bf3ef09afebcfdadebd66d
c71512f731fd9e9ca7c05d8caf4fef63
db922ea144a2a99edc7069fc6b6b7a6d
ac9eaccfdf60a6a50d4716ffbf012511
7a7477ffb6120c1e3cad044e4c00200c
a1ebab44ad99e97a96952bbd189e3bf7
dd7680148235776b83f5355eaac1259f
70caa351b1f68306876be12292762b19
86099251ea4e577fdbf732928eec0756
058a0092509d341f0925e20f644aaa4a
da879b282ad672a5e31d1a610285778a
6a890edae526e67dd6c9bcd6223417a3
da2a0dde23b5adefef9acf7a87ed6f8e
2579d9b9b7e3b3fd966cc2a03000a1a9
dbd53da111f226b0b10858b2875710ff
8f90c2c0cd35980297d60eb7125deb58
9dc38eda8768709a162b44a33930a532
a4a901a0dc6d79889c022cabd9fc29ec
6ce28a37396e727b71bbc09f789ae0e3
f6e9341216c403e94f1510238800be1c
390417a789210dfa96321640d17992d5
9ad5043d7d669fa52c7509625fb26c13
672e36c63b932d83918b7e0fc383d807
6d54c4849c4bc6de06642d676d361a83
a72e87753c0e3345a96c36cf53366711
3df822d0b5a0e0725fb4ab07498cb753
c54e30779317cf9e588f4edb9e00cf93
f81e314ddb5c1af7923a4ee967dec16e
df65cf7dff99ce1c1654175df3307aa0
407b5b65e2e9faba61be1587c7559ec1
55c3e203be9454bff685d68971a90cb9
ddbca7f89ff170377a94711799060ff0
8a490b5675cd2d76d8dac77387369a72
d6c0ee6878edbb2eedffda133d32e7d8
1e4576d9336f204c4f7ff416bdeba0df
8437795613b383c8000af8356ab9285a
b4ff7b38edb88b79af1787d922e6b39a
104528e67f01168e12cdac550fc43260
4b9406e27cf4332ffa2d78b1ef8b986f
ab19e26233537edc42843172293deb77
00dca835bb93708797a053a3b540db16
dd9c152f9965ed947dc6667ab4a2c545
d98e128bc2a462b7b7774b03e5d25078
a1cc525428bf22fa3068c8dbcb6d507c
d9a2585dc11272f9305784fb71c1fa31
57a07921b9a8e7ee6fe5421466ae15d7
7dd59150717878681976f811acbc116f
dc617075b90c59efb3ff3e8ac6788ce0
04b70b6985b638035014d2eb6a41391c
5dbd9956b4e15026ea2c06427e7d0a4d
0e746a512379ddd47519937491652176
02b24a492e792eabedff4746ba1b4240
acd613ac96a22ff3e98bbf76024fb1a3
526446e9c9267e344433d2b1e2a27950
f3a17cb9919d6d5e92af37f0a3f71575
c601a5a1a129488c4ae1ad7b55d9e243
afa5200cec6270b821245cc8daf80cfb
8e79997df7f235e4ef30553376382e53
b0b2041ea6601942970647e690430e08
de6370680ce959101117d44b1db00aef
fe5a5164a447a473bf41fe4fd7434078
00881545fa5768e4a94fadff8db5df49
d4ecf11a5188a1647488790fa32fc8c8
5373ae00e9fb34bfb30693fd7c533285
de10711b36f5657d4abb0ff40b53d3f8
df2dbb5f963100cde06a777e650dc8a3
68ea4fbac42c7376e18d61298fafb476
2eaf243bad4b1c22089e7654524f0e5a
dd770eb10539934eb711349d37220ac6
655a01ec9f936e275edd130fcc90f2d3

bfbcc08397c441d6498fa822cb7b268f
c36d05f428122b58cdaca3006c288323
dbbb88b6fbfcc7e0261a430d50339984
8badf2ca3eef55efb3043950b365729a
df50eeedf488e22e9f56c6431a24f468
78d232d7da0e3bc89defe62ab162fec6
a5f86063d20096f63db2b00d5e2d0e57
c4eb6552105da6a93696119c94e70110
f328f2cb570f41ff26913a0e796b1772
f67dafe155d76c37db9ce876cc1f4744
ddbf27194a9a1b92a2dc7c5c71c462fc
d648adb828179bf98a6c268a45f54b80
91d1699a1e3a904a0a1ba80dc9862cc5
2c7ce5faf0479757edc3e71eeadc11d1
2f2269743b9124e401a4bc4c5a27459d
480460d178d903757908844775b74c8c
29c2c5e121f4da789b634d414328b4c0
dc30d2ef4c5f58b8a4b93eaeda71f323
a498137fa180d601e6d529e4c9508747
a25ed53a41eefdbbf8325dec39f328fc
5d4250d1a8f53b731e4b79e119a5a489
0707bd13b03b79ade9a1821b98c1e71e
df28a2f40b24619d8d8d0c12a0c6cd45
04dc2cb4c2a9d985372d617f10861f7b
c3735f46eeca1b0bd28f5af55230dfc9
c3536da59887a2f66d388f5a26bdacd6
613d479da711a116d57be0f48ae17fdd
bd3213f6bfa3eb033acf48c4fbac9097
dd552eec031bb7cd0bf9b3e09384cddd
59cb24053e96010c4a5631999d3cec4b
01424db3e7c4c1cd59a78df29277e009
da3dfaa8aad75096dcbf5aa0c63c68e2
aa2b1e82d3063126319d80e4dbda40c0

Xtreme

00047fb702c1bd3b20d2254650df3768
012ef682f57c5b242736fdbf533216ce
03afa299cdd5f04260c3c802a08e151d
03ccd72bae83487eea6585340c1f54ad
053c17c285aafa964535325a00eb934b
053ea0f162ef3f20bc2c02342f33b9e2
055ea57c27cf7b47b41135ea6dc1443a
07b836b6dad2ed74478a5e4b4d0fd4da
0ac05f67abc4ec757c96858a865adf7c
0e0ecf06774b7bd961a2d8d97b37b6f4
0f21d1349842c69674b9b17004cb5c7e
0ff429394157e4fca331c5f08c91899e
12d358ff057a96c11687c5d09410595e
13009b63dee4675c842ace362235f1d7
13eccdcf26d5a0495a2f205c9bd68c40
14176a9a467b67ed29fdc9bfaa89fad5
163abb0c7ca93edd01f5854465cb64c4
17f3da74634e562e7fe19c6c4e4e2bc3
1834e956f576a05805ecb824b23b5f28
187cf4f5091f9838bcf8cf82df4ba6b7
1999232fd02d0c267f2dc9696b82ce86
1d48820868d7711fc0d65df99a3c2df3
1d6b0df54f051dbb4855d7c31d22729f
1fc7935affe50d60ef087bef38b2d4f1
1fcd2c9bc63632d6e23490b04768991f
24aeab6b0c333c6c75ce2fbd781db178
26b2868c417367221699b392fff022dd
281bc2f3049df79163377480656fc30e
29aa8d848bd540540cbd216a3d0567e8
29b3e377ffd6c479a96c97ea38dcd9ec
2ab89b6319ca4256d27f88c8f3c6ff05
2ae2b09f9a9eb4a4c981c43ef8642ef7
2af454cace8a047bc87d17b33151b71c
2c06eab6814c9789410285a3cd21d12e
2f33c24cb9dd37639ffccd7cac1e9dbf
30eb6483e143bd625ea1986fb0063514
32e73a98543f0f9ca9b2126f323c7e58
332704bce32a7d10aadca990c6fd24f4
33b7bee5818f8b2308170d9374428625
3448085381bc85de2363b3e7bb940838
34a4423bc864c829810b461e8d76f7d9
34e982abb0a2a4d835c3b453d35063a3
3592b29962a9ca40fafaac79b6460b93
3930398dc09233736af17faf7137a4fc
3a197ad72e1808a94b0b9d1124f3fbbd
3a48bd69cd6c8f484b578835ff347c3a
3ed74518ed6a875f533db40057be26f0
3ed813685d098c623ac97a5a50265647
40ee2641605669dde236ba26c9ddfaeb
40f93cd71fa289c174aba946dab07abc
43c89d639c5f1033aca75fff2f7d09e9
44cd04d0c681452741c77b330fe7f0ec
47d0f8e92839cbdefe66c54f06671692
4889f7ebdb03204d8b97de40211c9882
4bcf521c724edbb6184ce54a6efe11f3
4c3906ae023c569f351cd6d675c0f0ae
4c4a502dfe46b3cf242ee286915b3091
4d46918268e3e0a40e8893182e1916d0
4d85c0086a8add9dc27d1630b51b02be
4e76ad2139234f43e49d5bac6a909516
4e85e3a69be777f4043be63540f6e97d
5064f4d882e0f92061265de63e340527
509792c2d1ac577116c9f5dc419e1178
5206506397fddfcb2cdc639b155a5d63
5452d9f6da91f30d0147d52e587773fe
54d154b0c019ac1ce78557c14bd2ee29
5518ca3496c8e24b84c12b5e09780831
558185ee2311843dc571f8e8aa0cf186
558f9e2cd560c9fd3c274ac8e4c97778
580759da374049f50cd5bdd92235dfd9
5a23de6447a9294e4b00129649e42b6b
5b454b62e087143916f48f021f17f80c
5b86f52c5402ee5091a43d3b6c34f9b5
5d4aeede9c2477cb53024d454a5f19fb
607e598bb7d6a5804cacb9fce9da3757
64647b1a6ce44ccea4603b22d0c872a7
65568920b004c74ab94e23c57515016c
66136f40e9629e9db585474d75bcbee8
666952adabd976b1baf3f78f021aa9c4
6697a6265596585b0efac772fe869da3
679b19c199282a402e0297a0cd3b9823
6832a9e81de26c141ace88c3f9f53a12
68565d1c42475f7c060c746fe77579a7
697632f4dc1850e4cbeb36912a4c3044
6aa1ec3ed70945cd85b2a8dafe9b55b7
6b7bf65f0bdc849c6f7811e4189f474b
6f7165f69900b5abd505749fc69dfdc1
7173ff3ed650215eb618bfc9c6d9bd30
72b3bf916d9fc12ee4b46a9609e5cb35
7619e19d3d4aec9ab12142975d5cc177
76965fd607f94445c7904be4b4e33daf
76badb708d12455b3e06eb0b50010d2d
76cce82b75abb63957bf1ac1ab28e9db
7836d5f7f1e1ac4db4e9ac588f3c36c1
78b66a3d27096f4959332300e59132b4
78bf4a5176aa02f79868979ec92439a8
7a82d47220a53d201120fba89706aa7b
7c5ee1e690b31bbb9f971886e01bdbe9
7dbe90a5d8d065ebbf45efc40fff345b
7f114aaff8333cee561dcb815d7ba780
80651ef04c7323340af351d48761aada
807e0d36758d6b6bce5ac22f4c4de35d
822b89419badab03f62107dceec5c5cd
8280a34d5b0415223c659b7b49b2d32a
83079614d1e882f808d8542f6c03d9d4
834570ce6baec79e3b2b6b36b3928d0e
8a13703a1bcff8f68d20084337005143
8a954e5a8992dbbe835808ca7bcf121c
8b71efcbfe62a3c68723559b21da6320
8ba2cadd7821ca2b4259b21af776f513
8c2dd62748e9e86d37c6aeb04dc7f6e6
8c40f636d72a05e3079ff75e2bd81cce
8fd2ba061de501525e8d3c5a64f56273
9189169d74005760a091bfd127cd457b
92057608f2b49602efd4c6f3b738b819
92e6f9c4aea0c83f6f3f5e1a8fa35b07
96764ec96630c6ddb826ea63e0ac4dc9
96d5a3821427f57abb4934f3ee86354b
99bbb7003cf64e7ae3899415ea4c4ba3
9e0a499265fa583be8ec00382e33201a
9f828761585a7a3d9394153860e3f718
a03e8726d0d48087d51902d0c40b81b4
a3a4c632697c12ba4a0c3042867a1c6a
a4b4245c10bc99714b38fdda9cfe47e5
a5f4683e71d7b3790cd513abd97a5647
a721b97e2ec292968c23141171ed8439
a95aa8f30ca55435085536fa44267ebe
a9e364993af77022b908d22eae14c89f
ae786bf8360db97a33991eb4a0550ff4
aecf1e9fbb5a3c428a367f62ab52d39e
b064b4710ccdcd15987ca4ce9bdcc4b3
b113eaeeac8352f87b3610fc8f5e9230
b3c23f912778a946fd81806b5b8f66c9
b53db050b357feaed7437db4186f1665
b5aafce997968a48e61c4af5e2ebc2bb
b6e0674c89fbd2a5f6870fade56dcb59
b7c3620e2e6d44933196fe2dcc269057
b8417df837c5db0e029cf53d6629d9b6
b96786038d462389cd8b89194a42bff5
b96b2796f76bb75c9dbe37a05dd711e5
bb735ce094e3a9f28ee60a0acaa1501d
bd18e1c99caf6a1348a5646d8f711bc7
bd300ea0bbb0edf1e5aed2cdb058e8a7
bd8f9b665655bcb6dae2c0540d20feae
c0a4db45a13ad446cb465b8f98abcb7b
c19762025fd962fb27c07ac86424dc57
c27739bc50567c7726b47b6de95b569c
c2acdde0c6060f42e1f8b63a887fde1c
c49b7dd47d772ff6f808e9387de1dbab
c4eeeb97a1ba6517a33aa4f6a1a0cb47
c56520d5e3232acb596ec9211703fb19
c7b2fc8e9ce9af5db5b567aee900d81b
c9214f802a583cf5b78e89c319736e6d
c9f690ca0aa4f2656e2d008625ee6a5e
cbcb15a3814c2664af31a03659249a08
cd677bb61de2334c2195e114e5cfe9c0
ce46d3bc68edc03dc3885e472c851ea5
ce4a7098eb4d2618fe3dd6864031369b
ce7f4ad96f49f3deef0fa37c70e4e832
d408f71a26299473904dbe3dba137c76
d451cd1793ebab031f4041112104b9b7
d62044aa8e1f9fac97a1f45a193e8912
d9901c589eaa340eb2cfa6134424cf83
dd2e46b407eef8372fcd41cf521c8016
ddc9a13c0a5152d92086dc145516e2a6
de29344d8c9502add8e4c26d4b97f216
e13891ac63e18be50d4250c8e279ebe0
e24e3970516af0a878a01fe61e17a8a9
e2b1d139ac08fe865e11780cf4507af5
e33b3037962037b72fe8b7c9de4959bc
e366b06f0a9546b03fa67715b5fd737f
e388050c5f04c03e6a256da3ca301433
e62ac8c331028f256bfffae67c9e555b
ea5c7217a9039891bfcf67aad76e456e
ead53e2f497e040cce972afc6fc3a62e
ecea54c62a9e77799e2ff3ebeecf0430
ee3f30d5e47214a39315aa122a5b61d1
eee54d68e3482a87315ca3de13c2a976
efc4a39372d2db4fa21275ad4206cea0
efc5350a5f8d3cdf61f61f108a31b3c7
f07bd87f4ae1674ec3861d037aef1148
f21baefff256722dbabe77859522df98
f22e678501a4d9d1dcbe1e0ac9e3abb7
f2c2c57ff3f7e7b5849e134f352a5b2f
f35899c46def8ee218e59ad7ae38d015
f388549d7b7ba26fb6c58ba8d59ce260
f3e1836a47a2fb1780625811252940b6
f3eb7e7b75a0cfa9d1ad13b005b32313
f492570ce33486c2ff760dfbd566e73e
f51e48b2984bd1ca6b5eb04347c7be9e
f56157feb0056643827d34d093fc49b4
f67379246dae6b88600e920ed693dfd1
f7cc7946f3b9fdd073259b5ea50eaa3a
fb18fa5a3477953e9551560fe10cbe0f
fb8b3d6e284eb236f2b86d2853687323
fbbbb695075d9b10b4732be3ce418e65
fc29e524ee35f53f583019730574d7c0
fd0ad9aacf6d98ff37f286f5138df45b
fd86f140aa0539d26364e530a8d3f8b3
fdc2db15d85c679687ee325bbf80517e
febd97c9c08282ef198b48a274e0122e
fff2c8b832433e6181f7ddf1b93e79a1