Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Static Analysis for Malware Detection

Štěpán Dvořák

Supervisor: Ing. Jan Stiborek
May 2018

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Dvořák**	Jméno: **Štěpán**	Osobní číslo: **435012**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Otevřená informatika**		
Studijní obor:	**Softwarové systémy**		

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Detekce škodlivých souborů pomocí metod statické analýzy

Název bakalářské práce anglicky:

Static analysis for malware detection

Pokyny pro vypracování:

1. Review current methods for detection of malicious PE/PE32 files (executable Windows files) and identify their problems.
2. Analyze information in PE header and instructions and propose their numerical representation.
3. Propose method for classification of PE files.
4. Test proposed solution on a representative dataset and compare it with existing methods.

Seznam doporučené literatury:

[1] Nataraj, L., Karthikeyan, S., Jacob, G., & Manjunath, B. S. (2011). Malware Images?: Visualization and Automatic Classification. Proceedings of the 8th International Symposium on Visualization for Cyber Security - VizSec ?11, 1?7. http://doi.org/10.1145/2016904.2016908
[2] Raff, Edward, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. "Malware Detection by Eating a Whole EXE." arXiv preprint arXiv:1710.09435(2017).
[3] Oliva, Aude, and Antonio Torralba. "Modeling the shape of the scene: A holistic representation of the spatial envelope." International journal of computer vision 42.3 (2001): 145-175.
[4] Ahmadi, Mansour, et al. "Novel feature extraction, selection and fusion for effective malware family classification." Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. ACM, 2016.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Jan Stiborek, centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **31.01.2018** Termín odevzdání bakalářské práce: ______________

Platnost zadání bakalářské práce: **30.09.2019**

Ing. Jan Stiborek podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání — Podpis studenta

Acknowledgements

I would like to thank my supervisor Ing. Jan Stiborek for his valuable guidance and advice. He was always helpful and put on a brave face when my programs were crashing.

I am grateful to the CTA team at Cisco Systems, Inc. for providing me with the opportunity and resources to work on this project.

My thanks also go to my family and my girlfriend Tereza for all their support.

Declaration

Abstract

Malware detection becomes a necessity with the rising amount and power of cyber attacks. Nowadays, Windows operating system is the most widely used system, and for that reason, it is targeted the most by malware authors. We focus on malware detection in *Portable Executable* files since it is the most common file format used for spreading malware in Windows OS. The amount of malware is too large, so it is not possible to analyze all the samples by hand, even only running them would cost too much time and resources. Therefore, it is necessary to analyze binaries without actually executing them, i.e., perform *static analysis*. Many methods of static analysis have been proposed, but they lack comparison on the same real-world data. In this thesis, we provide a comparison of features extracted from raw byte code, PE header, and assembly code. We evaluate gain of each feature separately, and then select the best performing set of features and use them to train Gradient Boosting Tree. We tested our approach on a dataset of almost 900,000 binaries collected in the wild between November 2017 and January 2018. Our dataset is much larger than those usually used for research and contains a wide range of malware variants. We show that proposed approach provides sufficient results for deployment in real applications. Besides, we provide a thorough analysis of obtained results to understand where and why the classifier makes mistakes.

Keywords: static analysis, malware detection, portable executable

Supervisor: Ing. Jan Stiborek

Abstrakt

Se sílícími kyberútoky a jejich roustoucím množstvím se detekce škodlivých souborů stává nezbytnou. Operační systém Windows je v současnosti nejrozšířenějším systémem, a proto je také nejčastějším terčem útoků. Zaměřujeme se na detekci škodlivých souborů ve formátu Portable Executable, jelikož právě ty jsou nejčastěji zneužívány pro šíření virů ve Windows. Množství škodlivých souborů je příliš velké na to, aby bylo možné zkoumat soubory jednotlivě. Dokonce ani není možné soubory pouze spustit, protože by tento proces stál příliš mnoho času a prostředků. Z těchto důvodů je nutné zkoumat soubory bez samotného spuštění t.j. aplikovat statickou analýzu. Bylo publikováno mnoho metod statické analýzy, avšak chybí jejich porovnání na stejných reálných datech. V této práci porovnáváme příznaky získané z binárního kódu, PE hlavičky a assembler kódu. Vyhodnocujeme přínos každého z příznaků, tak abychom vybrali nejlepší množinu příznaků, na které natrénujeme Gradient Boosting Tree. Naše řešení jsme otestovali na 900 000 spustitelných souborech, které jsme shromáždili od listopadu 2017 do ledna 2018. Použitý dataset je mnohem větší než ty, které se běžně používají pro výzkum, a obsahuje širokou škálu škodlivých souborů. Námi navrhované řešení dosahuje výsledků nutných k použití v praxi. Navíc v naší v práci detailně analyzujeme získané výsledky a zkoumáme kde a proč náš klasifikátor chyboval.

Klíčová slova: statická analýza, detekce škodlivých souborů, portable executable

Překlad názvu: Detekce škodlivých souborů pomocí metod statické analýzy

Contents

Figures

Tables

Chapter 1

Introduction

The number of malware in the wild is constantly rising (see Figure 1.1), and AV-test reports over 250,000 new malicious programs every day. The total number of malware reached 700 million in 2017. Windows operating system is still targeted the most, and Portable Executable is the most common file format used for spreading malware [6]. Attacks performed by malware are becoming stronger and threaten even critical infrastructure of countries. In 2017, WannaCry ransomware infected over 230,000 Windows computers in over 150 countries. Victims of the attack include British healthcare system, several hospitals, the large telecommunication provider Telefonica, as well as the German railway company [30]. Developing effective malware detection systems that would detect and stop cyber attacks is crucial to protect our computers, privacy, businesses, and infrastructure.

The task of detecting malware is different from many other machine learning fields in many aspects. Among others, attackers write their programs in a way to evade detection. They are at the same time the source of data for creating a classifier. This situation is very challenging. Another important aspect of the anti-malware engine is its scalability. With the number of binaries on computers, the system must be able to handle a large number of coming files and classify each file as fast as possible. Furthermore, the need of making right decisions about files is another crucial aspect when detecting malware. Rising too many false alarms would make users turn off the system, but at the same time, we want to detect as much malware as possible.

Several data-mining methods were proposed in a literature [29] [11]. There are two types of malware analysis: *static* and *dynamic*. The difference is that during static analysis a binary is not executed. Static analysis is very challenging since we do not see the behavior of a file. Published methods extract information from PE header, raw binary code, or they go further and disassemble the binary to obtain assembly code. These approaches show promising results, but they are hard to compare since they were not tested on the same datasets.

In this work, we compare several methods used for malware detection. We create our custom dataset that is large enough to cover many malware

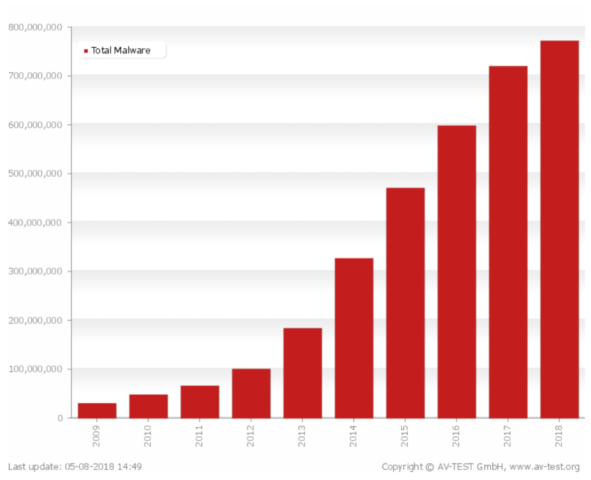


Figure 1.1: Total number of malicious programs registered by AV-test during the last 10 years. [6]

variants and simulates real-world scenario. We compare various features that are widely used for malware analysis, and find the best performing set. We further add a detailed analysis of obtained classification model, where we explain where and why the classifier makes mistakes. The resulting model is efficient and capable of handling large datasets while having satisfactory performance. Our work can serve as a baseline when building a malware detection model, and it provides a valuable introduction to the field of static analysis.

The thesis is organized as follows:

In Chapter 2 we explain the concept of Portable executable (PE) format. We focus on information that is valuable for malware detection, and we also mention some countermeasures against static analysis.

In Chapter 3 we review published work concerning malware detection. We summarize used methods and report their results. We add a section about machine learning classifiers and explain the model we used.

In Chapter 4 we formulate requirements on datasets used for malware detection. We survey public datasets and introduce our custom dataset, and explain metrics that are used for measuring the performance of classifiers.

In Chapter 5 we define our method starting with features we use. We give reasoning behind used features, explain how to select the most relevant ones and build a classifier on them.

In Chapter 6 we show results of particular used features, and we find the best performing set of features. We train a classification model on them and give a detailed analysis of the obtained model, and we further improve it.

We conclude this thesis in Chapter 7 and give suggestions for future work.

Chapter 2

Portable executable file

Malware comes in many different forms and types, but Portable Executable (PE) is the most common file format used for spreading malware [6]. Knowledge of PE files is necessary to understand how we can extract useful information for malware detection. Firstly, we give examples of files that are in PE format. Secondly, we explain the structure of PE file and explain its important parts. We include a section about assembly code that can be obtained from a binary. Finally, we explain packing and how it impacts static analysis.

2.1 PE file extension

PE format is used by:

- Executables (.exe)
  Almost every program that a user executes comes in the form of .exe file.
- Dynamic-link libraries (.dll)
  DLLs are widely used in Windows OS. They contain code and data that can be used by other programs.
- Other minor formats like FON Font files

Difference between DLLs and EXE files is negligible for our problem, so we treat them the same. DLLs are meant to be loaded by other programs so they export functions and data that can be loaded by other processes. When EXE file is executed, it resides in its own process and starts running its program. EXE file usually does not export any functions, but it can import other libraries.

2.2 PE file format

PE file contains all data and executable code as well as all necessary information for Windows loader. The loader gets information about sections of the file and allocates memory for them, resolves imported libraries of the program,

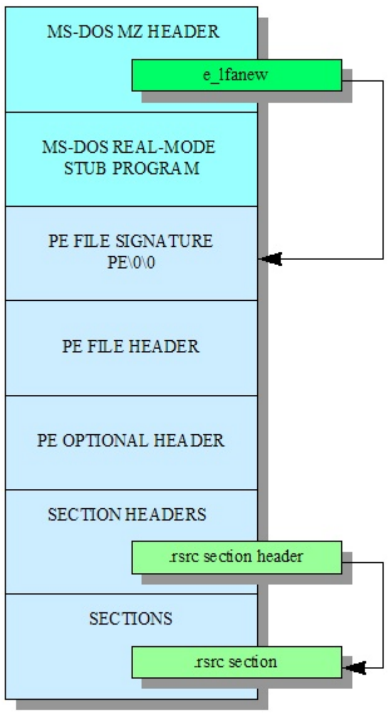


Figure 2.1: Structure of PE file [2].

and passes execution to the program. PE file has predefined structure so loaders can read it across all versions of Windows OS and execute the file properly. The structure is shown in Figure 2.1.

PE starts with *DOS header* usually consisting of a message "This program cannot be run in DOS mode." It should contain field e_lfanew that points to a beginning of *PE header*.

PE header has information about the whole file. Some of the entries in the PE header:

- Target machine
- Timestamp when the file was created
- Number of sections
- Size of Optional header

Optional header follows and gives more details about the file. Some of the entries in the Optional Header:

- Entry point address - where the program execution starts
- Size of code, Size of data
- Magic code representing architecture (PE32, PE64, ROM)

- Pointers to Data directories - addresses and sizes of tables like import table, resources table

Section table follows and gives information about each section. Each record in the table contains, among others, name of the section, virtual address and virtual size. Typical sections are: .text, .code (executable code); .rsrc (resources); .data, .idata (initialized data); .bss (uninitialized data). However, section names can be user-defined and there is no fixed set of names.

2.3 Import Address table

As we mentioned previously, there are data directories in the PE file. Table with imports is one of them, and it is a valuable source of information for an analyst because imports give us an idea about the functionality of a program. Authors of programs usually use libraries, so they do not have to implement all the functionality by themselves. Using libraries saves time and work during development. Furthermore, libraries already installed in the computer can be reused so they are not bundled with the binary and the binary is smaller.

Libraries are typically used for standard operations like writing to a file, accessing the internet, etc. We can use `CreateFile` function from `Kernel32` library for writing to a file instead of writing the code by ourselves. Other typical procedures like reading Windows registry, using Graphical User Interface (GUI), cryptography, and many others are done calling library functions.

Imported libraries are connected to the program by a process called *linking*. There are three types of linking:

- *Dynamic linking* - Libraries are resolved by OS when the program is executed. All the linked libraries are listed in the PE header.
- *Static linking* - The code of the linked library is copied into the executable. There is no information about static linking in the PE header.
- *Runtime linking* - Libraries are connected to the executable only when they are needed. These libraries can be resolved using functions `LoadLibrary` and `GetProcAddress` which may or may not be listed in the PE header.

Dynamic linking is the best option for a security analyst. Information about dynamically linked libraries is inside the PE header so by reading it we can at least partially understand the functionality of the program. For example, we can find out that a program uses GUI, or accesses the internet, just by looking at its imports.

Statically linked libraries can be discovered by digging deeper in the binary and matching the code with known libraries. Runtime linking is almost stealthy for the static analysis. We can discover libraries supposed to be linked in runtime by finding their name as a string in the binary. However, even that does not have to be possible if the binary is packed. An explanation of packing follows in Section 2.5.

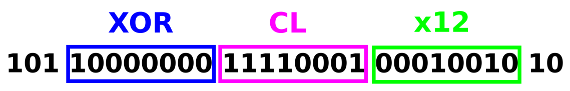


Figure 2.2: Machine code and its translation into assembly instruction "XOR CL 12".

2.4 Code section

Executable instructions (code) of a binary usually has an intended section (e.g., `.text`). Code data are usually read-only because they do not need to be modified during runtime. Instructions are in a binary in the form of machine code. The machine code is specific for given hardware and is hard to read. Therefore, there is a mapping between machine code and assembly code. Assembly code is more general and easier to understand. In this section, we explain what is assembly language and how to extract it.

2.4.1 Assembly language

Assembly language (ASM) is a low-level programming language used for giving instructions to the computer. ASM is specific for each architecture, e.g., x86, MIPS. There exists a mapping between assembly language and a machine code where the machine code is in a binary form and is directly interpreted by the hardware. We can think of the assembly code as a human-readable version of the machine code. A program called *assembler* transforms the assembly code into the machine code. The reverse process of converting the machine code into the assembly code is called disassembling and is performed by a *disassembler*.

Instructions of the assembly code are in the form of *operational codes (opcodes)*. An opcode represents a basic operation, e.g., `ADD` represents addition. The opcode is followed with zero or more operands. Operands can be constants, registers or addresses in memory. For example `XOR CL x12` performs logical XOR between a value in the register CL and constant 12 (hex). Machine code representation is shown in Figure 2.2.

2.4.2 Disassembling

A PE executable contains machine code which can generally be translated into assembly code instructions. It is necessary to locate the entry point of the binary to start disassembling. The entry point is an address in the memory where the program starts, i.e., the address of the first instruction. If we shift the address just by one bit, we can interpret the machine code as a completely different program. This fact is visible in the Figure 2.2. The translation starts at the entry point and continues based on the data found. Data and code can be mixed, and therefore the translation should be done with care. The code contains many GOTO instructions (jump, call, etc.) which affect the flow of the program. The code is usually divided into functions based on GOTO instructions.

Having the assembly code is a powerful tool for the analysis of a program. By reading or executing the code, we can understand each part of the program. Binaries can also use specific code constructs or opcodes that can be typical for some malware.

2.5 Packing

Packing generally takes an executable and transform it into another executable with the same functionality but different properties. It is mainly used for compressing and encrypting the binary and preventing its analysis. Even legitimate software defends against analysis because then it would be possible to copy it or easily change it. Packing of binaries is popular among malware writers because it provides effective protection against traditional detection techniques such as pattern matching. To understand why packing makes analysis difficult it is necessary to understand how packing works.

2.5.1 Unpacking stub

There are several techniques of packing. Generally, they transform an executable into another executable called *unpacking stub*. The stub is completely different program than the original one. Its only purpose is to make the original binary run. The unpacking stub must do three steps:

1. Unpack original executable into the memory
2. Resolve imports
3. Forward execution to the original entry point

The original executable is stored as an encrypted binary blob in the packed executable, so the stub needs to load it into the memory. When a binary is loaded into the memory, the unpacking stub must resolve all imports. Resolving can be done using one of the types of linking described in Section 2.3. Common approach is calling functions `LoadLibrary` and `GetProcAddress` and resolving all imports. These two functions are then in the PE header. Another approach is having all imports of the original program in the PE header of the unpacking stub. Then imports are resolved by the OS loader, and the stub does not have to do anything. However, the functionality is exposed which is something the malware authors do not want. The stealthiest option is performing runtime linking and finding even `LoadLibrary` and `GetProcAddress` functions on the computer and not exposing any imports at all.

2.5.2 Unpacking of binaries

If we analyzed a packed binary as is, then we analyze the unpacking stub and not the original binary. Therefore, unpacking of the binary should be performed to obtain the original one.

Firstly, it is necessary to detect that a binary is packed. A packed file usually has a high entropy of sections since it is compressed and possibly encrypted. Another way is finding signatures of the packers [19], e.g., packers can create sections with typical names, or append specific byte sequences.

Retrieving the original binary (unpacking) is a very complex problem. There are packers (e.g., UPX) that provide also decompressing procedure. This case is the easiest one, but the decompression tool is usually not available.

The most reliable unpacking procedure is running the unpacking stub until the execution is forwarded to the original binary (original entry point). However, there are many difficulties connected to this approach. Detection of the original entry point is one of the many challenging problems. Another problem is that the unpacking stub needs to be run and it can be potentially dangerous.

Generic unpackers are another way to go [19]. First, they detect the used packer and using this knowledge they try to reverse the packing method. This method requires a database of packers and a lot of effort in reversing the packing. There are many packers available, and attackers can even write their packers, so building a generic packer is a challenging task.

Unfortunately, there are currently no fully automated unpacking tools that would handle all kinds of packers. Packing a binary is a huge obstacle in the way of static analysis. Packing can hide useful information hidden in the binary,e.g., imports, strings (packed binary does not contain readable strings because it is compressed or encrypted), original sections; and can make disassembling impossible. Writing a custom packer evades detection by a generic unpacker and then running the binary for analysis is a necessity. However, addressing the unpacking is out of scope of this work.

Chapter 3

Related work

Malware detection is becoming more and more important since the number of malware is continuously rising and attacks are becoming more powerful [6]. Therefore, many researchers focus on finding a large-scale solution to prevent cyber attacks. Malware detection system needs to find a good set of features to represent a binary file, and appropriate classification model must be selected and trained. In this chapter, we review literature that addressed these problems.

3.1 Malware analysis

Malware is malicious software that intentionally harms a computer user. Malware can vary in its form (executable file, pdf file, etc.), and functions (trojan, spyware, adware, etc.). Detection of malware is essential for protecting users' data and privacy. Therefore malware detection systems are deployed to check files and mitigate their harm. Malware detection systems are fully automated because they must handle a large number of files coming every second and it is impossible to classify them by hand.

If we examine a binary without actually running it, then we talk about `static analysis`. When the sample is executed, we talk about `dynamic analysis`. We focus on static analysis in this work.

3.1.1 Static analysis

Classifying binaries using static analysis is a complicated task because we cannot directly see the behavior of an executable. Thus, we extract information from the binary structure (e.g., PE header). We can go further and disassemble binaries to obtain assembly code as explained in Section 2.4. By understanding the code, we can understand the binary and understand its purpose. However, automating this process is an ongoing challenge.

Schultz et al. [28] introduced data mining to start malware detection on a bigger scale. They used three types of features: *strings*, *byte sequences*, and *Portable executable features*. Strings are extracted as series of printable

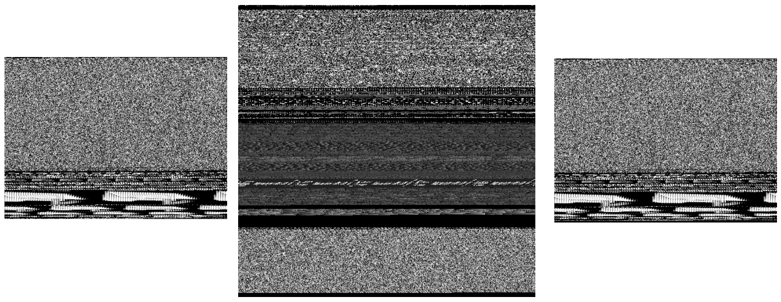


Figure 3.1: Binaries from Ramnit malware family transformed into images.

characters embedded in the binary. Byte sequences are non-overlapping sequences of n bytes. Portable executable features are extracted from the PE header. They include a list of imported libraries, list of library function calls, and a count of different function calls within a library. Their dataset consisted of 4,266 binaries, and they obtained 97.76% detection rate with 6.01% false positive rate (explained in Section 4.4).

Elovici et al. [9] used bytes 5-grams together with features from the PE header (e.g., machine type, file size, imported libraries, exported functions). They trained several classification models, and the best performing one was a decision tree trained on PE header features which they tested on 7,694 malicious files and 22,736 benign files. They obtained TPR 0.925 and FPR 0.035.

Kolter et. al [17] used *n-grams* instead of byte sequences. The difference is that n-grams are overlapping sequences. They selected the 500 most prevalent 4-grams out of all generated. Their features represent whether the n-gram is present in the binary. They tested their model on a dataset of 1971 benign executables and 1651 malicious executables. They obtained the area under ROC curve (AUC) of 0.996. They also compared several classification models and concluded that boosted decision trees provide the best detection performance.

Nataraj et al. proposed visualizing malware binaries as gray-scale images [21]. They take a binary as a vector and transform it into an image. The width of the image is derived from the length of the file. The proposed widths are in Table 3.1. Malware images show similarities between samples in the same family, and this fact is used for classifying binaries. Example of 3 images from Ramnit malware family is in Figure 3.1. Authors used tools for analyzing textures and extracted GIST features [23]. GIST features provide a low dimensional description of an image. K-NN classifier was trained using these features on a dataset of 9,458 binaries from malware 25 families. They performed multiclass classification and obtained 99.2% accuracy. They also tried adding benign executables to their dataset, and then their accuracy dropped to 98.1%. Authors pointed out that this method is vulnerable to adversaries that obfuscate the code to make texture features useless.

Raff et al. [25] proposed training a neural network on raw bytes. This

File size	Image width
<10 Kb	32
10 - 30 kB	64
30 - 60 kb	128
60 - 100 kB	256
100 - 200 kB	384
200 - 500 kB	512
500 - 1000 kB	768
>1000 kB	1024

Table 3.1: Image widths proposed by Nataraj et al. [21].

approach does not require researchers to manually design features. Instead, a convolutional network finds useful patterns in the binary. They compared several datasets where the biggest one used for training had 2 million samples (1,000,020 benign and 1,011,766 malicious). Training on the big dataset took a month on 8 GPUs. They tested their classifier on two datasets (77,349 and 65,821 files) and obtained AUC 98.2 and 98.1 respectively. They compared their network to n-gram ($n = 1$) approach and obtained AUC 93.4 and 97.0 respectively.

Microsoft Malware Classification Challenge [27] in 2015 provided dataset with 21,741 samples. Almost 400 researchers participated in the challenge, and other researchers have used this dataset since then too. Microsoft shared each sample as a binary and also output from Interactive Disassembler (IDA). IDA output contains assembly code.

Ahmadi et al. [5] participated in the challenge and achieved 99.8% accuracy. They used features both from raw binaries and IDA output. Their feature set consist of metadata features (file size, entry point address), section statistics (details in Section 5.1.3), n-grams of bytes and opcodes, image features, entropy, and many other features. Their framework is available online.

Winners of Kaggle competition [15] used n-grams of opcodes up to 4-grams, segment line counts, and pixel intensity of IDA output file transformed into an image. Those features contributed the most to the final solution, but they also used byte n-grams, dll calls, and some other features.

3.1.2 Dynamic analysis

When malware is executed for analysis, we talk about *dynamic analysis*. Dynamic analysis is a powerful tool for malware detection because a binary can potentially show its malicious behavior during its execution.

There are several challenges when performing dynamic analysis. Malware must be executed in a safe environment to prevent spreading. Dynamic analysis is usually performed in a specialized Virtual machine called *sandbox* which records the behavior of a binary (created files, network activity, etc.). Chen et al. [8] showed that malware changes its behavior in a sandbox, and therefore this environment should be designed with care. The result of dynamic analysis depends on a scenario of the experiment which must cover all situations in which the malware could be malicious (e.g., some malware only activates when browsing the internet). Another problem is that malware can start being malicious after a long time, so short analysis does not discover its maliciousness. Sandboxes nowadays can perform detailed analysis and are likely to discover the malicious behavior of binaries.

In conclusion, valuable malicious indicators can be obtained during dynamic analysis. We can see what files were created/deleted, network activity, etc. This information directly reveals the purpose of a file and show its potential maliciousness. However, malware equipped with anti-sandboxing techniques may remain undetected which limits the performance of this approach. Giving more details about dynamic analysis is out of scope of this work.

3.2 Classifiers

The goal of classification is creating a prediction model, that will assign a label to each vector in defined feature space. In this section, we define classification problems and continue by explaining Gradient boosting decision trees that we have used for our problem.

3.2.1 Classification definition

Let us define a general classification model. Let L be a set of data labels. In our case of binary classification:

$$L = \{\mathit{malicious}, \mathit{legitimate}\}\,.$$

We define classification model as a function f such that

$$f : \mathbb{R}^D \rightarrow L$$

where D is dimension of data.

Machine learning uses training data to build a model that can be used for classification. The model is found by minimizing chosen lost function. For train set T

$$T = \{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$$

$$\forall i = 1 \dots n : \mathbf{x}_i \in \mathbb{R}^D$$

$$\forall i = 1 \dots n : y_i \in L$$

where n is number of training samples. We minimize loss function:

$$l = \sum_{i=1}^{n} J(f(\mathbf{x}_i), y_i)$$

where J is arbitrary cost function, e.g., mean squared error. Classification models differ especially in f function and also in chosen J function.

The usual flow of creating classifier is as follows. Firstly, labeled data, which we want to classify, are described by features. These features should be descriptive enough to make the classification possible. A part of the dataset (called train set) is used during the training phase of the classifier where the prediction model is built. The model should be tuned to achieve maximal generalization to unseen data. Obtained model is then tested on data that were not part of the train set, and results are reported. Examples of classification models are Support Vector Machines, Random Forest, Neural Network.

3.2.2 Choosing a classifier

We want to use a dataset of binaries to train a model that will be used for classifying binaries as legitimate or malicious. Fernández-Delgado et al. [10] showed that *random forest* is the most likely to perform the best on classification problems. Random Forest is an ensemble method that constructs multiple decision trees, and the final decision is obtained by aggregation of results of the trees. Random Forest also returns the decision path which shows how the decision was made. It helps to interpret decisions of the forest and understand them. Another model based on Decision trees is called *Gradient Boosting Decision Trees*. It is built differently than random forest.

3.2.3 Gradient boosting decision trees

We explain Gradient boosting decision trees (GBDT) in this section. Boosting solves the need to classify *difficult* samples right because it puts more weight on them during training. The tree is built by sequentially adding weak classifiers. GBDT is a version of Gradient Boosting Machine where all weak classifiers are regression trees. Algorithm 1 shows how the tree is built. Model is initialized with a constant value in step 1. Then we sequentially add M weak learners in step 2. A model for each class is built separately (using softmax function guarantees that probability of all classes sums to 1). In each step, a weak learner fits negative *gradients* of residuals of the classifier from the previous step. A tree partitions space into J_m disjoint regions, and the tree predicts a constant value in each region R_{jkm}. This value is multiplied by γ_{jkm} obtained by line search in step (iii). The resulting model is updated in step (iv). The tree outputs a score for each class, and the class with the highest score is then predicted for a given sample.

3.2.4 LightGBM

LightGBM [16] is an implementation of gradient boosting decision tree. It is suitable for large datasets because it is highly efficient and fast. It can be used for many problems including regression and classification.

LightGBM introduces algorithms to implement gradient boosting efficiently while preserving accuracy. Gradient-based One-Side Sampling is a way to weight samples during training. Data instances with larger gradients

Algorithm 1 Gradient Boosting Tree Algorithm [14]

```
1. Initialize $f_{k,0}(x) = \text{argmin}_\gamma \sum_{i=1}^{N} L(y_{ik}, \gamma), k = 1, 2, \ldots, K$.
2. For $m = 1, 2, \ldots, M$
    a. for $k = 1, 2, \ldots, K$
        (i) Compute
            $$r_{ikm} = -\left[\frac{\partial L(y_{ik}, f(x_i))}{\partial f(x_i)}\right]_{f=f_{k,m-1}}$$
        (ii) Fit a regression tree to the targets $r_{ikm}$ giving terminal regions
             $R_{jkm}, j = 1, 2, \ldots, J_m$.
        (iii) For $j = 1, 2 \ldots, J_m$ compute
            $$\gamma_{jkm} = \underset{\gamma}{\text{argmin}} \sum_{x_i \in R_{jkm}} L(y_{ik}, f_{k,m-1}(x_i) + \gamma)$$
        (iv) Update $f_{km}(x) = f_{k,m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jkm} I(x \in R_{jkm})$.
3. Output $\hat{f}_k(x) = f_{kM}(x), k = 1, 2, \ldots, K$
```

contribute more to the information gain (used for building a tree). This fact is used to keep instances with high gradients during training and randomly sample instances with smaller gradients.

LightGBM also implements Exclusive Feature Bundling for reducing the number of features. Many features are exclusive (they do not take nonzero values simultaneously) and therefore can be bundled together. This fact is used to lower the dimension of features. The resulting classifier trains much faster while having almost the same accuracy.

In the field of malware detection, a model is required to handle large datasets and minimize the number of misclassifications. LightGBM is very effective and capable of consuming big data. Therefore, we use LightGBM as a classifier in our work.

Chapter 4

Dataset

The number of malware in the wild is continuously rising, and AV-test reports over 250,000 new malicious programs every day with Windows OS being targeted the most [6]. Malware samples often change their code and adopt more advanced evasion techniques, so their detection is becoming harder. The proper dataset should reflect trends in malware and simulate real-world situations. In this section we describe requirements on datasets of binaries, briefly overview current public datasets and then describe the one we created.

4.1 Requierements on dataset

Several requirements naturally arise when finding a dataset:

- *Dataset should cover as many malware families as possible.*
  Malware can be divided into families where the family has the same author or uses the same vulnerabilities of systems. Samples from the same family can also use similar code constructs, parts of the code, or libraries. These similarities between samples in a family can help to discover new variants of malware that were just slightly modified. By including more families, we also cover more variants of malicious behavior because families usually have different behavior.

- *Dataset should be as large as possible.*
  Large dataset generally helps in building a classifier because it prevents overfitting and helps to generalize to unseen samples. Getting a large dataset would not be a problem but the problem is labeling the samples. It is not possible to manually process a large dataset, and therefore labels need to be extracted from various sources (e.g., from scans of antivirus engines).

- *Dataset should simulate real-world situations.*
  The usual situation is that binaries are collected for a given period and then a model is created. This model is then used for evaluation of samples coming in the following period of time. Public datasets lack this

property and therefore are not suitable for evaluating real-world malware detectors.

4.2 Public datasets

Unfortunately, there are not many public datasets of binaries. Researchers usually create their own collection of malware from virus sharing services and legitimate samples from their organizations or clean installation of Windows OS. Unfortunately, results from researchers are not comparable because they lack common benchmarking dataset.

The newest widely used dataset is from Microsoft Kaggle competition [27]. However, competitors showed that it is easy to get nearly 96% accuracy by using only three simple features. This dataset contains around 20,000 samples from only nine malware families. The number of samples, as well as families, is highly insufficient.

Public datasets usually contain less than 50,000 samples [27][21]. This number does not represent the volume of files in the wild, and it has only a small fraction of malware families. Consequently, it is hard to evaluate classification of new unseen malware families. Datasets are also aging very fast. Antivirus engines are continually improving so malware authors must improve malware to evade detection, e.g., by finding new vulnerabilities or by improving its defense against analysis. Therefore, it is necessary to keep datasets up to date.

4.3 Custom dataset

We created our own dataset that meets requirements previously described. It is large enough to create model generalizing to unseen data; covers hundreds of malware families; and was collected in consecutive periods of time to simulate real-world situation.

We use binaries collected in the wild between November 26th 2017 and January 1st 2018. There are more than 900,000 samples that we were able to label. Usual source of labels is VirusTotal (VT) [4]. Researchers upload binaries there, and they are checked by antivirus engines (AVs), then a sample is considered legitimate if no AV finds it malicious. We used similar service like VT called *Titanium Cloud* [26].

Titanium Cloud (TC) provides file reputation service and returns labels of three categories - *malicious, legitimate, suspicious.* TC uses many features to make this decision. Unfortunately, authors do not provide detailed information about their engine but only basic information. They make the decision using whitelists and blacklists of files, PTI (Proactive Threat Indicators), antivirus-detections and possibly others. PTIs are based on the behavior of a binary, resources embedded inside, etc. We resolved all our samples in Titanium Cloud two weeks after the last day of our test dataset and used the result as

	legitimate	malicious	suspicious
train set	515092	42624	19858
test set	310435	29304	12065
total	825527	71928	31923

Table 4.1: Labels distribution in our dataset.

labels for training and testing. We were able to label 929,378 samples out of all collected. Labels distribution is summarized in Table 4.1.

The task of classifying our dataset is very challenging. Classes are very unbalanced since we have much more legitimate samples than malicious ones. There is also possible overlap between suspicious class and the other two. Therefore, we leave out suspicious class and use only legitimate and malicious. We split the dataset based on the time when samples occurred because we wanted to simulate the real-world scenario. We use binaries collected between November 26th 2017 and January 4th 2018 as a train set, and the test set contains binaries from 21 following days.

4.4 Metrics

When building a model, it is important to measure its performance numerically. We want to compare several models and select the best one based on our criteria. In case of malware detection, our priorities are:

1. Catching as much malware as we can.
2. Minimizing the number of false alarms.

The first criterion expresses that we want to classify malware samples as malicious. Otherwise, we would not discover any threats and our system would be useless. The simple solution would be to classify all samples as malicious, but our system would be useless as it would generate a large number of false alarms.

The second criterion requires the number of legitimate samples classified as malicious to be minimal. If we raise false alarms for too many files, users will turn off the malware detection system. We use standard metrics to express our requirements, their description follows.

4.4.1 Confusion matrix

The confusion matrix is a table showing the performance of supervised learning algorithm. Our classification model requires each data instance to belong exactly to one class, and the model predicts these class labels. Confusion matrix shows what predictions were made on particular classes. Example of the confusion matrix is in Figure 4.1. Confusion matrix C is such that $C_{i,j}$ is

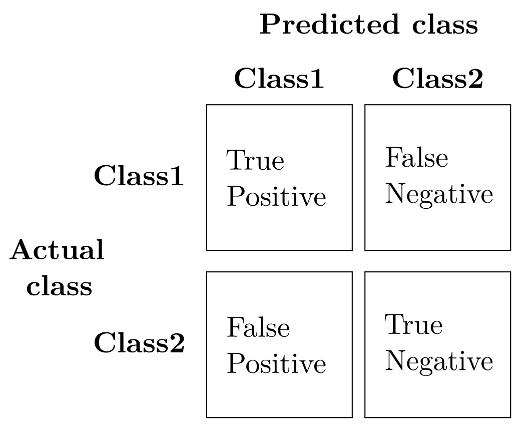


Figure 4.1: Structure of confusion matrix.

equal to the number of data instances of the group i that were predicted to be in the group j. In binary classification task:

- $C_{0,0}$ are true positives (TPs) - malicious samples classified correctly
- $C_{0,1}$ are false negatives (FNs) - malicious samples classified as legitimate
- $C_{1,0}$ are false positives (FPs) - legitimate samples classified as malicious
- $C_{1,1}$ are true negatives (TNs) - legitimate samples classified correctly

We further normalize the matrix by dividing each element by sum of its row.

4.4.2 True positive rate, False positive rate

True positive rate (TPR), also called *Recall* or *Detection rate*, is a standard metric used for measuring performance of a model. It is defined as

$$\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}.$$

It tells the ratio of malicious samples that were recognized.

False positive rate (FPR) is defined as

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}.$$

It tells us the ratio of legitimate samples that were classified as malicious. Minimizing this value is important since we do not want to rise too many false alarms.

Another standard metric is *accuracy* (ACC) defined as

$$\mathrm{ACC} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{FN} + \mathrm{FP} + \mathrm{TN}}.$$

However, accuracy is not suitable for our problem because our classes are highly imbalanced. Classifying legitimate samples right would cause high accuracy, but it would not reflect misclassification of malicious samples much. For example, if we had 95% legitimate samples and 5% malicious samples in our dataset. If we classify all samples as legitimate, we get accuracy 95% in this setup. This number sounds great, but the classifier is in fact useless. However, we get TPR 0% and FPR 0%. These numbers tell us much more about the performance.

To conclude, we evaluate our model using TPR because it shows the detection rate, FPR because it reports the rate of false alarms, and we add the confusion matrix to make our statistics complete.

Chapter 5

Method

Firstly, we need to find a representation of raw binary code in PE files, and express it mathematically. This representation must describe the binary well and capture potential maliciousness of the file.

In this Chapter, we describe features that we use to model PE files. Since the dimension of obtained feature space is high, we explain how we select the most relevant features. In the last section, we describe the configuration of the classifier that we use.

5.1 Features

The first set of features, we describe, models the raw binary code. These features are easy to get, and their extraction is fast. The second set of features models PE header of the binary. These features are well structured and usually give us valuable insight into the file. Finally, we disassemble the files and model the assembly code. Disassembling is a time-consuming and computationally expensive operation, but it reveals functionality of a binary. We need to express all the features as vectors and concatenate them to obtain a single vector describing the whole binary in the form of $\mathbf{x} \in \mathbb{R}^D$ where D is the number of features.

We use four ways to represent the features:

- Bag-of-words features [14] - we count the number of occurrences of given key in a file (e.g., imports).
- Hashed features [31] - hash of each key is used as an index to the array where the number of occurrences of given key is put. We hash each key to 4 different equally-sized arrays with different hashing seeds to reduce the number of collisions. Generally, features with a higher number of values are hashed to reduce their dimension. Generally, features with a number of possible values too large for bag-of-words representation are hashed to reduce their dimension (e.g., strings).
- Dense features - continuous features are represented simply by its value (e.g., size of a file).

Feature type	Features
Dense	number of resources, number of sections, mean size of resources, is signed, extreme import indicator, file size, entry point, haralick features, local binary patterns, entropy window, size of assembly file, number of opcodes
Bag of words	imported libraries, packer, machine, section type, resource lang, resource type, resource locale, certificate serial, special symbols, aop onegrams, aop onegrams normalized, opcode onegrams, registers, byte onegrams
Histogram	size per section type, size of section, entropy of section, section size per section, section properties, data define, registers normalized, opcode onegrams normalized,
Hashed	exports, imported functions, strings, tokenized strings, aop bigrams, aop trigrams

Table 5.1: Overview how each feature is represented.

- Histogram features - we create equi-depth histogram [24] with 10 bins, and add 1 to the bin corresponding to the feature value (e.g., entropy of sections).

Table 5.1 summarizes how we represent each particular feature. The description of our features follows.

5.1.1 Raw binary features

Raw binary features are extracted from the raw code. An advantage of this approach is that there is no other intelligence needed. We do not need to understand the structure of the PE file, but we simply use the bytes. These features can be hard to interpret because they only show general information about the file. The list of used features and their description follows:

- File size
- Entropy
  We compute the entropy of the binary code. Entropy is associated with the amount of disorder in a system. Higher entropy means a higher level of disorder. Therefore, measuring entropy can help to detect packed (compressed, encrypted) executables because they have high entropy. Entropy is computed using Shannon's formula:

$$e = -\sum_{i=0}^{255} p(i) \log_2 p(i)$$

  where $p(i)$ is the probability of appearing of byte i. The probability is estimated within the window. We do not compute the entropy of the

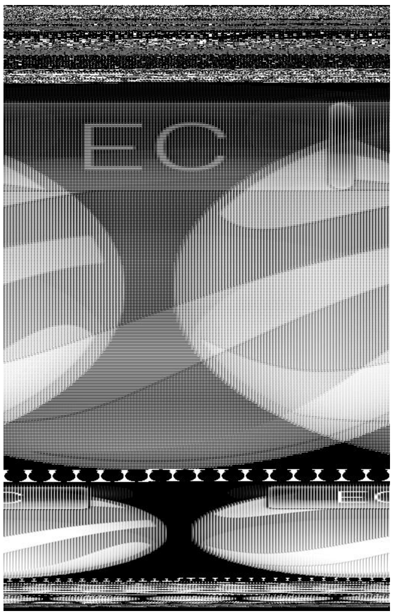


Figure 5.1: PE file transformed into an image.

whole file, but we rather use windows of 10,000 non-overlapping bytes. There can be only some sections encrypted (i.e., with higher entropy), and the rest can be not packed, windows find packed data across the whole file. Then we use statistical measures of obtained entropy windows such as mean, variance and quartiles.

- Image features
  The binary is transformed into an image as suggested in [21]. An example of a binary image is shown in Figure 5.1. The example is Google Earth executable so that we can see its logo in the resources. Binaries within a malware family have similar images [21] and image features should discover this fact. The similarity can be found for example in the resource section where icons are located.

  We extract *local binary patterns* (LBP) [22] and *haralick features* [13], which are both used for texture analysis, using mahotas [1] library.

  In local binary patterns, an image is first transformed by sliding a fixed-sized window over each pixel in the image. Pixels with lower intensity than the central pixel of the window are marked, and the window is encoded into an integer number. Histogram of these numbers is then used as a feature. LBP describe local patches, and they find interest regions of images (edges, lines, corners, etc.).

  To get Haralick features, we create a co-occurrence matrix of pixels. Element i, j of the matrix corresponds to a number of times pixel of intensity i is adjacent to a pixel with intensity j in an image. Statistics of the matrix are computed and used as features. There are 13 statistics in total, including correlation, entropy, contrast, variance. Haralick features

.net	ms visual c#	ms visual c++	pklite32	armadillo	upx	ms visual basic	ms visual c	bobsoft	install shield
5705	4149	3815	1337	841	737	346	297	104	77

Table 5.2: Prevalence of packers in 30,000 random samples.

i386	amd64	thumb	armnt	0xaa64	arm	ia64	sh3	mipsfpu	sh4
25591	4201	75	56	50	15	6	4	1	1

Table 5.3: Prevalence of machines in 30,000 random samples.

model spatial interactions between pixels.

- Byte n-grams
  N-gram is a continuous sequence of n bytes from a given byte sequence where byte has value in range 0-255. N-grams can help us to discover usage of specific bytes and specific patterns in a binary. They can learn from all types of sections (PE header, code section, data section, imports) reasonably [32]. We used only 1-grams since n-grams with $n > 1$ are computationally too expensive.

- Strings
  We find strings in the file as a consecutive sequence of more than 8 printable characters. Strings are often embedded inside of a binary. There can be attacker's messages for a victim, names of used libraries, etc. There is a lot of noise in strings, and more effort should be invested in cleaning the data, but we leave it for future work and use the strings without preprocessing. We compared using the whole strings and tokenized strings because there can be single words indicating maliciousness of a file. We used hashing trick to represent strings [31] because there are thousands of strings and we need to reduce their dimension.

5.1.2 PE header features

We use Cisco's internal tool called `Pe-tool` to parse PE header and to obtain useful information about binaries. This information reveals nature of the binary and gives us a valuable insight into the purpose of the binary as explained in Chapter 2. Other tools can be used to parse the headers, and they produce the same or very similar results.

We list used features and also give examples of values from randomly selected 30,000 samples to get better picture about the features:

- Imported libraries
  Names of dynamically linked libraries are extracted, e.g., "KERNEL32.dll". We can get a better picture of the behavior of a binary when we look at imports. Imports can show, for example, if a binary uses files, connects

DllCanUnloadNow	518	BdDestroyObject	95
DllGetClassObject	515	BdCreateObject	95
DllRegisterServer	479	____CPPdebugHook	66
DllUnregisterServer	472	qt_plugin_instance	63
DllMain	109	Initialize	61

Table 5.4: Prevalence of exports in 30,000 random samples.

kernel32	13282	shell32	3818
user32	8857	gdi32	3453
advapi32	8707	oleaut32	3291
ole32	5475	comctl32	2207
mscoree	4699	version	2028

Table 5.5: Prevalence of imported libraries in 30,000 random samples.

to the internet, uses GUI, uses the registry, cryptography, etc. Some malware families can also use specific libraries instead of the usual ones. The ten most prevalent imported libraries from 30,000 random samples are listed in Table 5.5. We also detect unusual imports (name longer than threshold 60) and use its presence as a feature.

- Imported functions
  The reasoning behind this feature is the same as for imported libraries but we get fine-grained information by using names of imported functions. We represent this feature as a bag-of-words where the key is "library name_function name". The ten most prevalent imported functions are available in Table 5.6.

- Exported functions
  PE files (especially DLLs) export functions so other modules (especially EXEs) can use them. Exports give us insight into the library and reveal what functionality the binary exports. The ten most prevalent exports of 30,000 random samples are listed in Table 5.4. For example, four most prevalent exported functions ('DllCanUnloadNow', 'DllGetClassObject', 'DllRegisterServer', 'DllUnregisterServer') indicate that a program implements COM server.

- Packer
  Many times binaries are packed either to compress their size or to prevent analysis. We detect the used packer by matching signatures. Malware families can use specific packers and detection of the packer can help detect the whole family. The ten most prevalent packers in 30,000 random samples are shown in Table 5.2.

- Machine
  There is a difference between binaries across platforms (e.g., i386 vs. amd64), and properties of binaries can depend on the platform. The

kernel32_GetProcAddress	10418	kernel32_TerminateProcess	9083
kernel32_GetLastError	10128	kernel32_WriteFile	8974
kernel32_GetCurrentProcess	9860	kernel32_MultiByteToWideChar	8974
kernel32_CloseHandle	9708	kernel32_UnhandledExceptionFilter	8796
kernel32_ExitProcess	9622	kernel32_WideCharToMultiByte	8785

Table 5.6: Prevalence of imported functions in 30,000 random samples. Names are in the form of "library name_function name".

most prevalent machines in 30,000 random samples are overviewed in Table 5.3. We can see that the biggest portion of samples is designated for i386, but there are many binaries for amd64 too.

- Section properties
  As we described in Chapter 2, binaries consist of sections, and these sections have specific properties that can help during classification. They have various names, sizes, and entropies. These properties can be common in malware families. We extract names of sections and use the size and entropy of each section as a feature. Pe-tool returns following types of sections: text, native, packed, or encrypted. The type of a section is derived from its entropy. We get a type of each section and use its size and entropy as features. Another feature is count of section types in a file and we add one more feature which is a number of sections in the binary.

- Resources
  Each binary has a resource section that can contain icons, menus, etc. We extract following statistics about resources: count, mean of size, languages, types, locale. Resources can help identify binaries (e.g., same programs differing only in version have typically the same icons).

- Signature
  Binaries can be digitally signed to validate the file and verify it has not been tampered with. The signature depends on a content of a file, and therefore it is possible to check its validity. However, many binaries are not signed, or malware authors can create their own certificates. We check whether the binary is signed and use the result as a feature. Serial of the certificate (identifier) is used as a feature as well.

5.1.3 Assembly features

It is possible to overcome detection based on bytes of a binary using obfuscation [20]. Therefore we disassemble binaries to obtain assembly code. Disassembling PE file is challenging task because malicious, and even legitimate software uses various prevention techniques where packing is one of the biggest issues as described in Section 2.5. However, assembly code is a valuable source of information and is widely used for malware static analysis [11].

We used Retdec [7], an open-source decompiler capable of transforming a binary into a high-level language like C or Python. We do not necessarily need that, so we use an intermediate output of Retdec in the form of assembly code. Retdec also tries to unpack the binary using either generic unpacker or UPX unpacker [19]. The problem is that Retdec cannot handle all binaries. In our dataset, 73% binaries could be disassembled properly. The rest was packed with a packer that is not supported or was compiled for a 64-bit system (currently not supported by Retdec). The output of Retdec is a list of sections containing functions or data all together in a text file, an example of the output is in Figure 5.2. The list of used features follows:

- Assembly file size
- Opcodes
  Opcodes (operational codes) are assembly instructions describing operations to be performed by CPU. We use opcode 1-grams as a feature. This feature detects unusual opcodes usage (can be used to evade detection), or other unusualities can be detected. We extracted 1,550 distinct opcodes from the train set.
- Abstract opcodes
  We transform operands of the opcode to a new representation we call *abstract opcode* (aop). Operands can be registers, constants or addresses in memory as explained in Section 2.4. Examples of transformation are shown in Table 5.7. We use n-grams with $n \in \{1, 2, 3\}$ as features. We also use relative frequency of onegrams in the file as a feature. A total number of 2,627 abstract opcodes was extracted from the train set.
- Registers
  Registers are built-in units in a CPU to store intermediate values and thus make computations faster. A frequency of used registres can help to assign a binary to a malware family. We use register name as features. We also use relative frequency of registers in the file as a feature.
- Section properties [5]
  PE files consist of sections that contain code and data. These sections can have various names, and there is no fixed set of them. Usage of specific names of sections can help to detect some malware families, so we extract statistics of sections in the assembly file produced by Retdec. We consider

  .text, .data, .bss, .rdata, .edata, .idata, .rsrc, .tls, .reloc

  as known sections, others are considered unknown. We count a number of known sections in a file. We also count lines that belong to each section in a file with assembly code. Then we compute ratios between unknown and known sections, particular sections and all sections, number of lines in known and unknown sections.

Original opcode	Abstract opcode
adc eax, 0x4029c400	add register constant
push esi	push register
mov eax, dword ptr [0x40601c]	mov register memory

Table 5.7: Transforming opcode to abstract opcode.

- Data define properties [5]
  Some samples do not contain any imports and instead consist only of `dd`, `db`, and `dw` instructions. These instructions declare data locations that can have assigned value and can be referenced later (similar to variables). These instructions help to initialize data of the program on the run, and therefore it can be a good sign of packing. We count a number of occurrences of these opcodes and compute ratio to all opcodes. `dd` opcode can have a different number of operands, and therefore we count separately dd instructions with 1, 4, 5, 6 parameters and again compute ratio to the total number of opcodes and also to the number of `dd` instructions.
- Symbols
  We count number of occurrences of specific symbols: -, +, *, [,], ?, @. These symbols may reveal indirect calls and can be a sign of detection evasion. Indirect calls can serve for dynamic library loading, so libraries are not discovered easily.

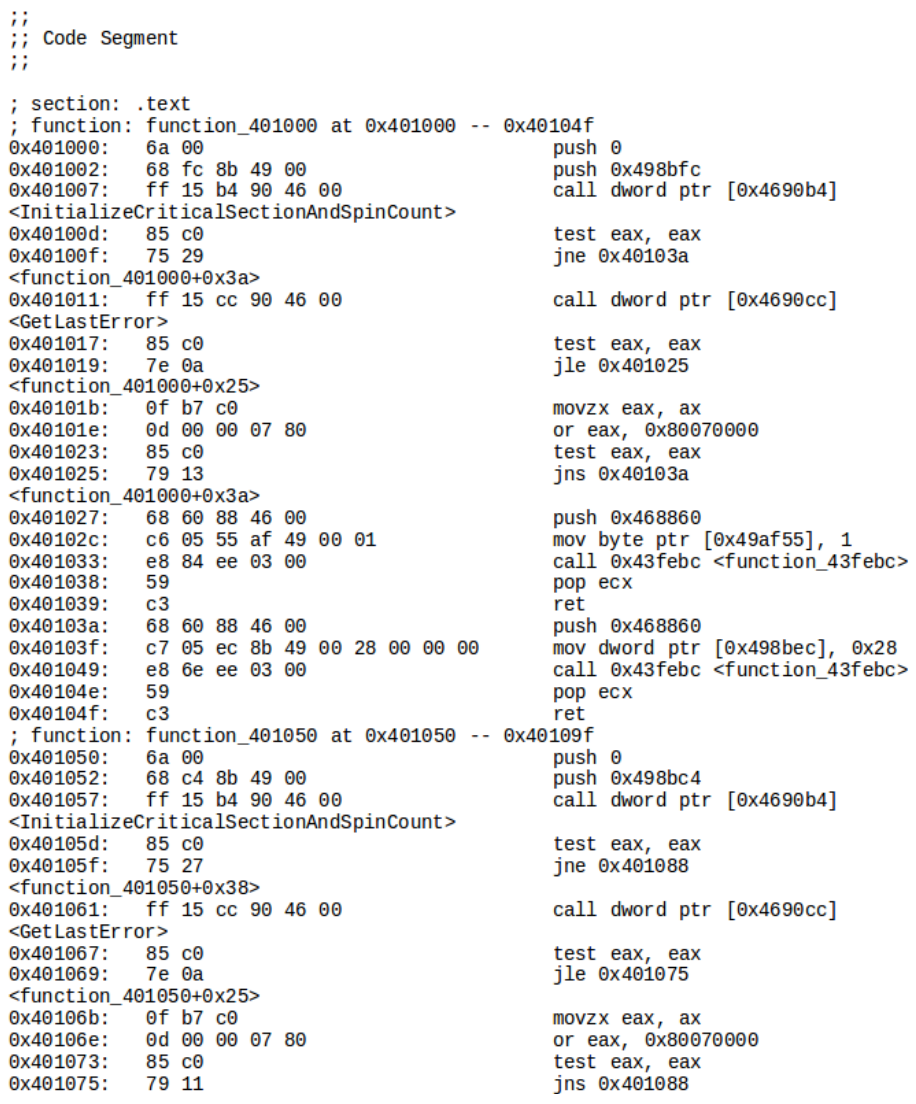
5.2 Feature selection

Previously, we described all the features used in our model. The dimension of combined feature space is $1,276,759$. Features with the highest dimension are those represented with hashing trick [31]: exports, import functions, strings, assembly bigrams, assembly trigrams. The dimension is too high to be computationally feasible. Therefore we use feature selection algorithm to select 10,000 the most relevant features. The model is then reasonably small and can be trained fast.

Shabtai et al. [29] reviewed feature selection method used for malware detection. They mention Document Frequency, Information Gain, Gain Ratio and Fisher Score. Computing Fisher score is easy and has linear complexity. More sophisticated methods are difficult to use because of the size of data. Since Fisher score provides an optimal balance between speed and performance, we decided to apply it to our data. A detailed description of this technique is given in [12]. Fisher score is defined as:

$$score = \frac{|\bar{x}_{positive} - \bar{x}_{negative}|}{\sqrt{s^2{}_{positive}} + \sqrt{s^2{}_{negative}}}$$

where $\bar{x}$ is sample mean and s^2 is sample variance. It computes the score of each feature as a difference between its distribution of positive and negative



```
;;
;; Code Segment
;;

; section: .text
; function: function_401000 at 0x401000 -- 0x40104f
0x401000:   6a 00                          push 0
0x401002:   68 fc 8b 49 00                 push 0x498bfc
0x401007:   ff 15 b4 90 46 00              call dword ptr [0x4690b4]
<InitializeCriticalSectionAndSpinCount>
0x40100d:   85 c0                          test eax, eax
0x40100f:   75 29                          jne 0x40103a
<function_401000+0x3a>
0x401011:   ff 15 cc 90 46 00              call dword ptr [0x4690cc]
<GetLastError>
0x401017:   85 c0                          test eax, eax
0x401019:   7e 0a                          jle 0x401025
<function_401000+0x25>
0x40101b:   0f b7 c0                       movzx eax, ax
0x40101e:   0d 00 00 07 80                 or eax, 0x80070000
0x401023:   85 c0                          test eax, eax
0x401025:   79 13                          jns 0x40103a
<function_401000+0x3a>
0x401027:   68 60 88 46 00                 push 0x468860
0x40102c:   c6 05 55 af 49 00 01           mov byte ptr [0x49af55], 1
0x401033:   e8 84 ee 03 00                 call 0x43febc <function_43febc>
0x401038:   59                             pop ecx
0x401039:   c3                             ret
0x40103a:   68 60 88 46 00                 push 0x468860
0x40103f:   c7 05 ec 8b 49 00 28 00 00 00  mov dword ptr [0x498bec], 0x28
0x401049:   e8 6e ee 03 00                 call 0x43febc <function_43febc>
0x40104e:   59                             pop ecx
0x40104f:   c3                             ret
; function: function_401050 at 0x401050 -- 0x40109f
0x401050:   6a 00                          push 0
0x401052:   68 c4 8b 49 00                 push 0x498bc4
0x401057:   ff 15 b4 90 46 00              call dword ptr [0x4690b4]
<InitializeCriticalSectionAndSpinCount>
0x40105d:   85 c0                          test eax, eax
0x40105f:   75 27                          jne 0x401088
<function_401050+0x38>
0x401061:   ff 15 cc 90 46 00              call dword ptr [0x4690cc]
<GetLastError>
0x401067:   85 c0                          test eax, eax
0x401069:   7e 0a                          jle 0x401075
<function_401050+0x25>
0x40106b:   0f b7 c0                       movzx eax, ax
0x40106e:   0d 00 00 07 80                 or eax, 0x80070000
0x401073:   85 c0                          test eax, eax
0x401075:   79 11                          jns 0x401088
```

Figure 5.2: Output of Retdec disassembler.

class. The difference is expressed in terms of mean and variance. The reasoning of this method is clearly visible in Figure 5.3. The higher the distance between means the better, and smaller variances are preferred. The score for each feature is computed on data from the train set and 10,000 features with the highest score are selected.

5.3 Classifier

Our goal is to train a model that for a given binary produces a label (malicious or legitimate). Previously we described how we represent each binary by a vector of dimension 10,000. We propose to use LightGBM [16] as a classification model because it is suitable for large datasets and provides interpretability of the model. We want to find out which features give the

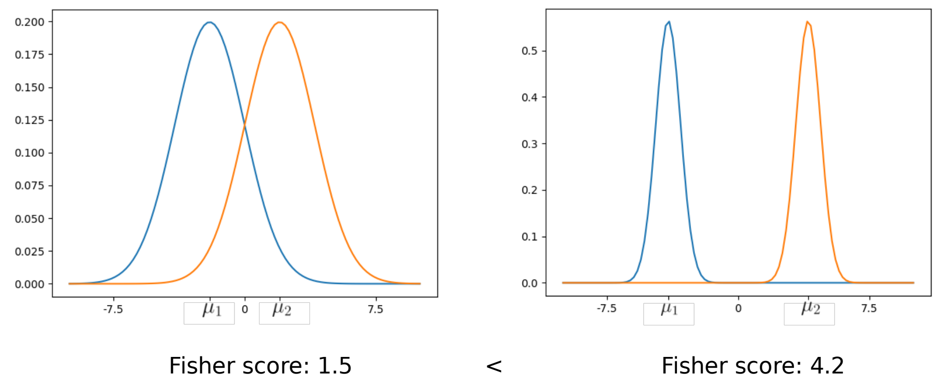


Figure 5.3: Example of two pairs of normal distributions with different mean and variance. Fisher score is computed and the pair on the right with higher difference between means and smaller variance is preferred.

best performance, and therefore we train a model with the same configuration on different sets of features and compare results. When we find the best set of features, we use it to tune the classifier. We use LightGBM version 2.1.0 and its default parameters. The only parameters that we changed are:

- n_estimators=200
- num_leaves=1024

Chapter 6

Results

We trained several models on features described in Section 5.1 using dataset described in Chapter 4. Results are reported in the form of True Positive Rate (TPR) and False Positive Rate (FPR) as described in Section 4.4. Our positive class corresponds to the malicious class and negative class to legitimate.

Firstly, we compare results on different sets of features and select the set having the best performance. Secondly, we analyze the model trained on the best set of features and optimize its hyperparameters to further improve its performance.

6.1 Feature results

We want to compare only the performance of individual groups of features, so the configuration of the classifier is fixed, and we change only feature sets. We try several combinations of features to obtain the best performing model. All results are in Table 6.1, TPR are compared in Figure 6.2, FPR in Figure 6.1.

6.1.1 Models Comparison

A model used by many researchers based on PE header performed well on our dataset but was not sufficient. The FPR 0.00171 is the highest compared to other models and therefore should be combined with other features. TPR 0.85579 shows good ability to recognize malicious samples.

Better in FPR (0.00077) but worse in TPR (0.83552) is the model using image features. This model, despite the limitation of the unnatural transformation of a binary into an image, showed comparably good results. It can be caused by recognition of legitimate resources in a file as samples from the same families have very similar images [21]. This fact can cause easier classification too. It should be further investigated in future work.

The model with features extracted only from raw binary shows better performance than the one with PE header features. This result is surprising because we expected structured information to be more discriminative. In the

	TPR	FPR
PE header	0.85579	0.00171
Images	0.83552	0.00077
Strings	0.82576	0.00107
Tokenized strings	0.82320	0.00098
Raw binary features	0.85422	0.00072
PE header + Raw binary features	0.86258	0.00066
PE header + Raw binary features + Assembly features	0.87363	0.00061

Table 6.1: Results of classification on different sets of features. 10,000 features were selected in all cases. True positive rate (TPR) and false positive rate (FPR) are reported. Positive class corresponds to malicious, negative to legitimate.

raw binary features model, 9978 strings were selected alongside with haralick features and entropy. The model with haralick features and entropy shows better ability to recognize malicious samples compared to the model only with strings. It can be caused by recognizing packers based on entropy, or by discovering similarities in the resource section modeled by image features.

Model using raw binary features was mainly based on strings. Therefore, we evaluated the performance of strings separately. Results indicate that strings are a good indicator of maliciousness but provide lower TPR. The problem is that packed binaries do not usually contain any readable strings.

We also used strings after tokenization because there may be only single words indicating the maliciousness of a file. We can see that with TPR 0.82576 for strings and 0.82320 for tokenized strings, the impact of tokenization is negligible. It can be caused by selecting strings corresponding to imports that are not changed by tokenization. Another problem is that we hash strings using $4 \cdot 2^{16}$ values. When strings are tokenized, much more of them are mapped on the same value due to collisions in hashing which can diminish the improvement gained by more accurate tokens.

When all features extracted from raw binary and PE header are used together, we obtain a model improving performance compared to the model without features from the header. It already shows a good ability to recognize malicious binaries (TPR 0.86258 with low FPR 0.00066) and the extraction of these features is much less complicated than disassembling.

Note that we do not evaluate a model with only assembly features in this section. It does not make sense on the full dataset because 27% of binaries could not be disassembled.

Even though not all binaries were disassembled, we use assembly features together with all others and mark them as missing values. Missing values are handled differently during training by the classifier [18]. When we used all features together with assembly features, we obtained a model performing the best on our dataset. TPR 0.87363 is the highest while FPR 0.00061 the

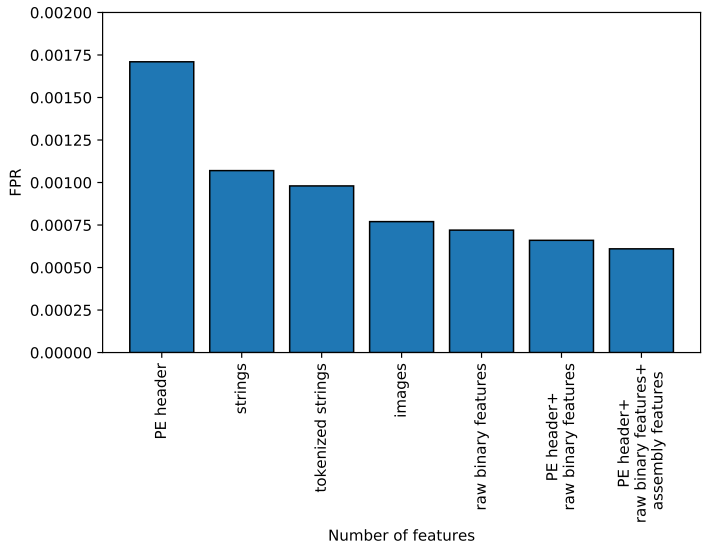


Figure 6.1: FPR of models trained on different sets of features.

lowest. Assembly trigrams were selected in feature selection replacing more than half of strings compared to the model without assembly features. Being satisfied with the results we must further admit that expectations of assembly features were a bit higher. The extraction of these features is time-consuming, and the gain should be higher. However, more sophisticated features can be developed. We take into account only the prevalence of opcodes and some other basic statistics. We do not try to understand the program and discover its true functionality. Another problem is that the model is very dependent on opcodes found. An attacker can insert redundant opcodes to obfuscate the code and then our features based on opcodes do not work. It is also questionable whether samples without disassembled features should be evaluated together. In this model, half of the features are dependent on disassembling, so their absence harms the overall model. We evaluate model using only assembly features in Section 6.2.

6.2 Only disassembled classification

We use Retdec [7] for disassembling binaries, but it fails on some of them. Detailed statistics of disassembling are listed in Table 6.2. We can see that most of the malicious binaries were disassembled successfully compared to only 71.6% of disassembled legitimate binaries. It suggests that legitimate software uses more sophisticated way of protection against disassembling or it can simply mean that Retdec does not support used packers or architecture.

We used all samples that have been disassembled to create a new dataset. Using this dataset, we can evaluate the real gain of assembly features. Details of this dataset are summarized in Table 6.3. We evaluated gain of each group

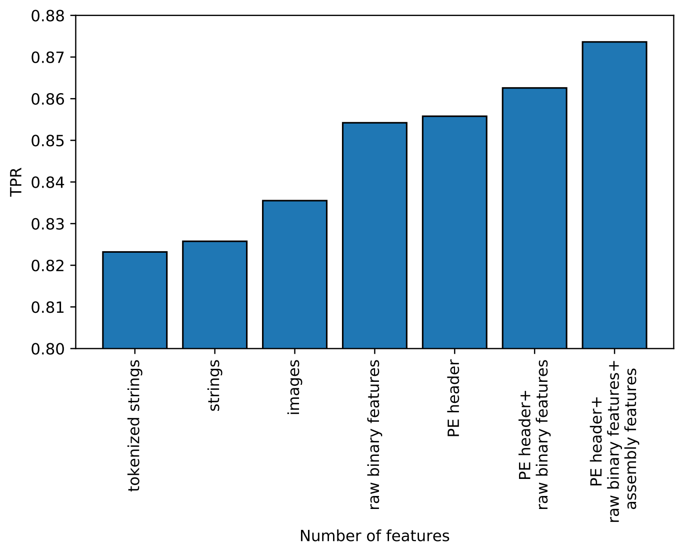


Figure 6.2: TPR of models trained on different sets of features.

	Total	Disassembled
malicious	71928	64110 (89.1%)
legitimate	825527	590326 (71.6%)

Table 6.2: Disassembling statistics.

of features. All results are summarized in Table 6.4.

We can see that PE header features give results similar to the dataset of all binaries. FPR 0.00201 is higher compared to other features. PE header features combined with raw binary features result in higher TPR 0.86366 while decreasing FPR significantly to 0.00068. Assembly features used by themselves show quite poor performance with TPR 0.77350. When all features are combined, we obtained a model with TPR 0.87406 and FPR 0.00065 which is similar to the performance on the full dataset.

6.3 Tuning the classifier

In all previous experiments, the classifier had the same configuration, and we have found the best set of features. We tune hyperparameters of the classifier in this section to further improve its performance.

In malware detection, our primary goal is minimizing FPR. Therefore, we select the best performing model based on FPR while relatively maintaining high TPR. We evaluated each parameter using 5-fold cross-validation [14]. In each step, we evaluate each setting of given parameter and select the one with the best performance, and continue in tuning other parameters. We

	malicious	legitimate
train set	38145	375376
test set	25965	214950
total	64110	590326

Table 6.3: Labels distribution in dataset of disassembled binaries.

	TPR	FPR
PE header	0.85165	0.00201
PE header + raw binary features	0.86366	0.00068
Assembly features	0.77350	0.00094
PE header + raw binary features + assembly features	0.87406	0.00065

Table 6.4: Results of classification on different sets of features on the dataset of disassembled binaries. 10,000 features were selected in all cases. True positives (TPs), false positives (FPs) are reported. Positive class corresponds to malicious, negative to legitimate.

repeat this procedure until no further improvement is encountered.

LightGBM has many tunable parameters, and we list parameters recommended for tuning and values that we tried:

- num_leaves $\in \{256, 512, 1024, 2048, \}$
- min_data_in_leaf $\in \{10, 20, 30, \}$
- max_depth $\in \{-1, 5, 30, 50\}$
- bagging_fraction $\in \{0.8, 0.5\}$ and bagging_freq $\in \{0, 1, 3\}$
- max_bin $\in \{512, 1024, 2048, 4096\}$
- learning_rate $\in \{0.001, 0.1, 1\}$ and num_iterations $\in \{100, 200\}$
- lambda_l2 $\in \{0, 0.1, 0.5, 1\}$

Changing most of the parameters harmed the performance, or caused only a very little change in the performance. We ended up changing only two parameters:

- max_bin=2048
- num_leaves=512

This setup results in average TPR of 0.87599 and FPR 0.00065 which is the best performance presented in this thesis.

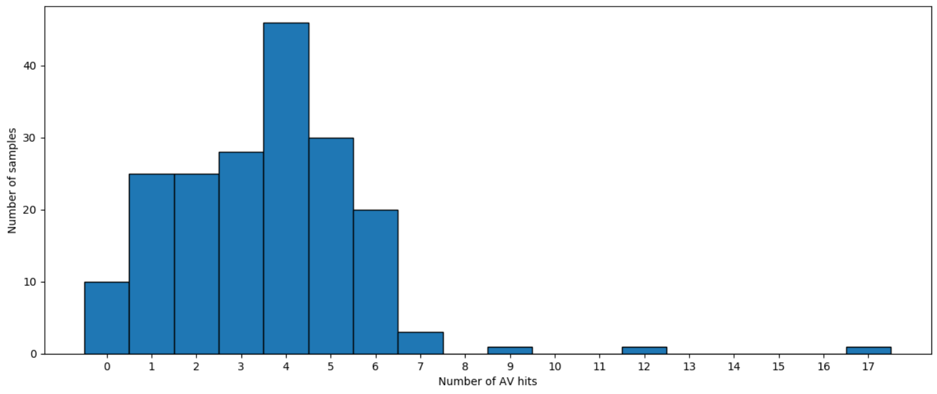


Figure 6.3: Histogram of antivirus hits for false positive samples.

6.4 The optimal model analysis

We have found out that the best performing model is the one using all the features we designed. We analyze which features are the most important in the model, so we better understand how the predictions are made. Then we analyze incorrectly classified samples to see where and why the model makes mistakes. We resolved samples in antivirus engines to obtain more detailed information. Antivirus engines return label of a sample which can contain a type of malware or malware family (e.g., Eset NOD32: Win32/Sality virus). If an antivirus engine classifies a binary as malicious we call it a *hit*.

6.4.1 False positives (FP)

Since we are interested in further minimizing FPs of the model, we looked at legitimate samples incorrectly classified as malicious. We checked their antivirus hits and discovered that many of them have some hits even from established antivirus engines. Histogram of antivirus hits for FPs is shown in Figure 6.3.

We tried to query all FPs of our model again after 2 months. We found out that out of 190 legitimate samples that we classified as malicious 15 changed its label to malicious and 4 to suspicious. There were samples with only 3 hits that turned into malicious as well as with 17 hits, so we cannot easily set a threshold on hits to decide that the binary is malicious. Of all 19 samples that changed the label, 8 were PUA, 6 Trojans, 3 Adwares, 2 Downloaders.

Some FPs are hard to classify even for human experts because we found samples that do activity similar to malicious samples. Those can be toolbar extensions that check typed URLs, software for tracking employees, applications tracking mouse movement and keystrokes.

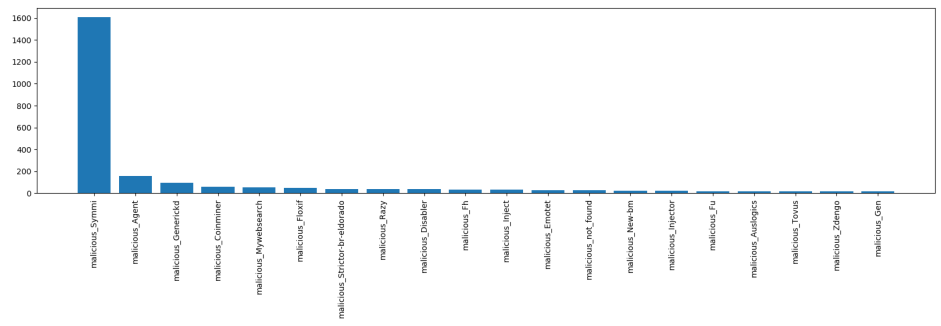


Figure 6.4: Malware families with the highest number of false negatives.

6.4.2 False negatives (FN)

Our classifier did not recognize many malicious samples, so we investigated them too. The most misclassified families are shown in Figure 6.4. The figure shows that by far the biggest unrecognized family was `malicious_Symmi`. Additionally, we checked antivirus hits of FN samples, and a histogram is in Figure 6.5. There are some samples with 0 hits, and many samples have only a few hits, so their maliciousness is questionable. There is a big spike in the histogram on number 4. It is caused by Symmi family where most of the samples have 4 hits. Many false negative samples do not have hits from established antivirus engines (like Eset, Avast, McAfee, etc.), but they have hits from engines that generally produce more false positives. Many times binaries are not matched with any malware signature, but rather have negative file reputation (e.g., Symantec-online outputs WS.Reputation.1). These files do not have to be necessarily malicious.

We manually analyzed some FN samples. There were cases when the binary was packed, so our raw binary features did not work well, since packed binaries do not have readable strings, and our model uses them a lot. These samples often could not be disassembled because of packing, so our assembly features did not work either. There were also samples hidden in Inno setup (Windows installer), and we did not catch them because the real program was hidden behind the installer.

We queried all FNs of our dataset again after 2 months. We found out that out of 3,703 malicious samples that we classified as legitimate 49 changed their label to legitimate and 8 to suspicious.

In further sections, we analyze the most prevalent misclassified families:

Symmi

It is a family with the most misclassified samples in our dataset. According to Symantec report [3], it is Trojan that opens a backdoor and may download files. There were only 66 training samples in our train set but 1,708 in the test set. We classified 1,611 as legitimate and 97 as malicious. We tried to

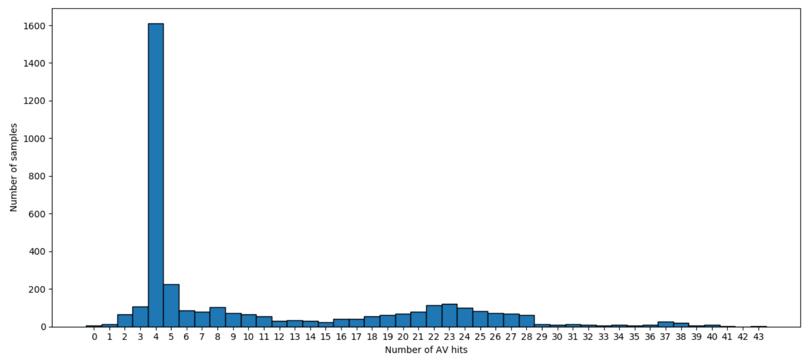


Figure 6.5: Histogram of antivirus hits for false negative samples.

	malicious	legitimate
malicious	0.92419	0.07581
legitimate	0.00061	0.99939

Table 6.5: Confusion matrix for a model trained on all features, computed without Symmi malware family.

run some of these samples to see what they do, but all of the selected samples were somehow corrupted and could not be run. We found out that 1,579 out of 1,592 FNs have 0 imports. We did not find a runnable sample between those without imports, and Pe-tool returned an error code on them. We can conclude that most of those binaries are corrupted, but there may be some that are really false negatives. If we compute results without Symmi family, we get results in Table 6.5. TPR is then significantly improved to 0.92419.

Agent

It is the second most misclassified family in our dataset. There were 1,425 samples in the train set and 932 in the test set where we classified 155 of them as legitimate. We analyzed some of these samples manually. Binaries ran without any problems. We could see their malicious behavior (e.g., opening ports to listen to attacker's commands). We focused on samples that were labeled as legitimate. These binaries were usually packed (encrypted) and had only a few imports. Our model is heavily based on imports (either from PE header or found as strings) and their absence helps to evade detection. There were also samples hidden in Inno setup, so we analyzed the setup tool instead of the actual binary.

Many samples also contained statically linked libraries, so the code of libraries is embedded in the binary code. When we search for strings inside the binary, we also collected strings from imported binaries. Other features are affected by static linking as well (e.g., entropy).

	malicious	legitimate
malicious	0.86115	0.13885
legitimate	0.00034	0.99966

Table 6.6: Confusion matrix for all features (raw binary + pe header + assembly) with threshold on malicious probability 0.9

6.4.3 Generalization

The ability of the model to generalize to previously unseen samples and families is crucial for malware detection models. We evaluate this ability in this section.

Each sample gets a family label from Titanium Cloud [26]. Titanium Cloud does not provide details how those labels are obtained. They are probably extracted from results that antivirus engines return. Those labels are sometimes noisy because there can be more names for the same family (e.g., 4e660515-eldorado and 4e75f222-eldorado).

There are 1031 malware families in the train set and 836 in the test set. There are 588 families that are present only in the train set while 393 families are only in the test set. Out of all 393 previously unseen families we had at least one true positive sample in 100 families, and 88 malware families have detection rate 100%.

6.4.4 Thresholding

We noticed that in some cases, the classifier was very uncertain about maliciousness of a binary and returned probability slightly in favor of the malicious class. The uncertainty led us to thresholding of probability needed to mark sample as malicious. This approach should minimize the number of false positives because we will say that a binary is malicious only when the model is very sure about it.

We say that a sample is malicious if its probability of being malicious (returned by the classifier) is higher than a certain threshold. Otherwise, we assign the sample to the legitimate class. Example for threshold 0.9 is shown in Table 6.6. This model improves FPR from 0.00061 to 0.00034, but TPR dropped from 0.8736 to 0.86115.

6.4.5 Feature importances

It is important to understand how the classifier makes its decisions. Decisions depend on used features, so we want to know which features are more important in the model. Generally, it is possible to extract feature importances from decision trees. LightGBM returns two types of feature importance: *split* (number of times a feature is used in a model), *gain* (total gains of splits that use a given feature). If a feature is valuable, then the model will likely select it more often, and the gain of the feature will be higher. We plotted obtained

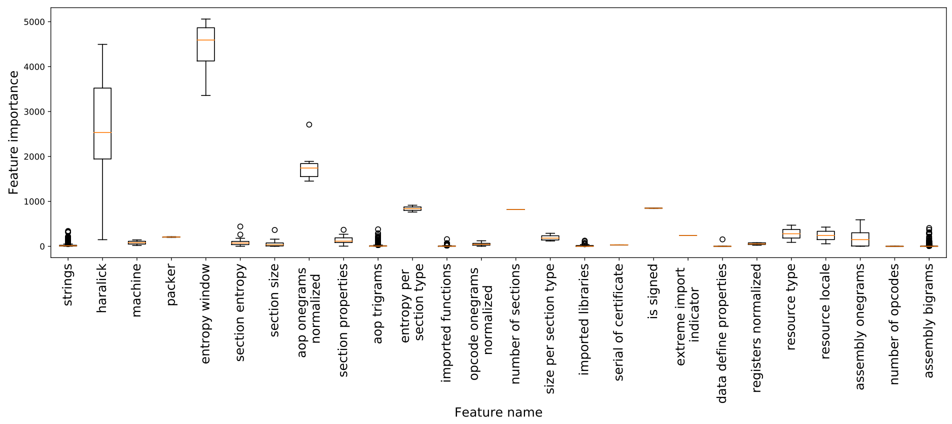


Figure 6.6: Feature importances (*split* parameter) of the best set of features on full dataset. Feature importance is given by a number of times a feature is used in a model.

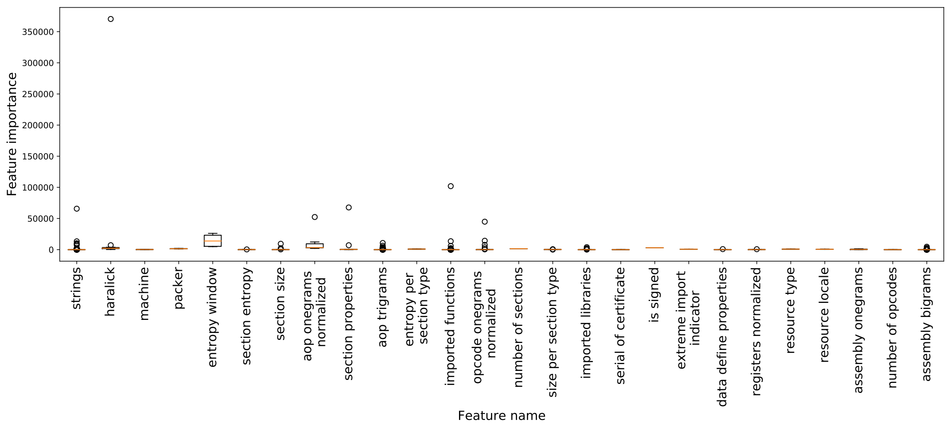


Figure 6.7: Feature importances (*gain* parameter) of the best set of features on full dataset. Feature importance is given by total gains of splits that use a feature.

importances: split in Figure 6.6, gain in Figure 6.7.

We can see that entropy from sliding window is considered the best feature. It is interesting that entropy of particular sections returned by Pe-tool is not rated as high. Haralick features have high variance in feature importances, and it should be investigated which ones are rated higher and why. List of features with the highest *split* feature importance is given in Table 6.7. This table shows especially the importance of entropy and Haralick features.

We further checked some features and their most important values are listed in Table 6.8. There are more features belonging to Assembly bigrams, trigrams and strings because we used hashing. Hashing collisions cause that more values are mapped to the same index. In case of strings, there are a lot of values mapped to the same index, and it should be checked whether increasing the number of hashed values changes performance.

Entropy Q3	5059	Haralick	3745	Assembly onegram normalized	2709	Haralick	2090
Entropy Q1	4802	Haralick	3705	Haralick	2535	Assembly onegram normalized	1891
Haralick	4496	Entropy mean	3359	Haralick	2534	Haralick	1798
Entropy median	4382	Haralick	3339	Haralick	2523	Assembly onegram normalized	1792
Haralick	3857	Haralick	2799	Haralick	2251	Assembly onegram normalized	1742

Table 6.7: Features with the highest importances given by LightGBM classifier. "Q1" and "Q3" mean first respectively third quartile.

6.5 Selecting more features

During our experiments, we always selected 10,000 features based on Fisher score. In this section, we compare performance when we select a different number of features. Increasing the number of features gives the model more features that can be used to discover malicious/legitimate behavior. Some features may be useful, but they were not selected during the feature selection procedure because their score is too low. Selecting more features gives more space even for low scoring features. However, increasing the size of the model comes at a price of time needed for training and memory usage.

We selected 10, 100, 1000, 2000, 5000, 10000, 30000, 50000, and 80000 features, and we compared their performance and time needed for training. We used Amazon EC2 instance m5.12xlarge with 48 CPUs and 192GB memory for training all the models. We trained each model 5 times and averaged obtained results. Times needed for training are plotted in Figure 6.8. We can see that the training time grows linearly with the number of used features. For example, training a model on 10000 features takes 556 seconds in average, whereas model on 80000 features trains for 4197 seconds.

We examine how TPR and FPR change with increasing number of features. Figure 6.9 shows TPR, and Figure 6.10 FPR. We can see that TPR is almost constant from 1000 to 30000 features, and a little increase comes with 80000 features. The model with 80000 has 0.87411 TPR on average, and the model with 10000 has TPR 0.87236 for comparison. FPR is decreasing with rising number of features. However, the change between 1000 and 5000 features is negligible. The model trained on 50000 features has the lowest FPR 0.00053.

Results indicate that more features provide better results in classification, but the time needed for training grows linearly with the number of features. With the rising number of features also comes higher memory usage, and

Feature name	Feature importance	Feature key
AOP onegram normalized	2709	add memory register
AOP onegram normalized	1891	pop register
AOP onegram normalized	1792	push constant
AOP onegram normalized	1742	call constant
AOP onegram normalized	1634	mov register memory
AOP onegram normalized	1470	call memory
AOP onegram normalized	1451	push register
Entropy per section type	914	native
Entropy per section type	762	packed
Section entropy	440	.rsrc
Resource locale	427	en_US
Packer	213	pklite32
Import library	126	comctl32
AOP bigrams	407	push constant + push register jp constant + rol memory register bswap register + cmp register constant
AOP trigrams	378	push register + push constant + push constant

Strings	341	(only some) getcommandline; chartStyle;
		COMODO RSA Certification Authority;
		GetCurrentProcessId; set_IsMonitorOn

Table 6.8: Selected features and their importance in the classification model.

80000 features were the maximum on the used machine. Choosing 10000 features seems optimal because the training takes only 556 seconds on average, TPR and TPR are very similar to models with more features.

6.6 Comparison with previous work

Several approaches were proposed in the literature as we mentioned in Chapter 3. We used features from the majority of papers we surveyed, but we altered them to need a minimum of human work. In our approach, we do not need to manually select values of opcodes, imports, or any other feature; but we represent and select the features automatically based on our current data. The goal of our work was comparing feature sets, and therefore we compare features used in other papers with our final set of features. We neglect other differences in methods, so we test only gain of particular features.

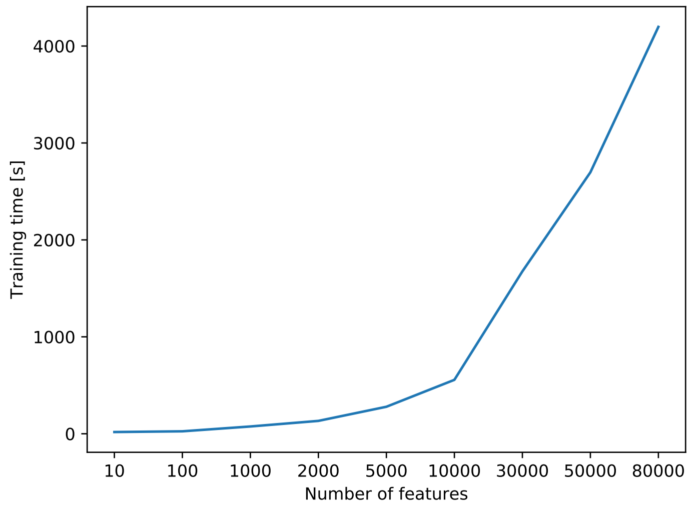


Figure 6.8: Training time dependency on number of used features.

We use the same method of feature selection as described in Section 5.2, and we train the classifier described in Section 5.3. We compare two papers with our model:

- Elovici et al.[9] used features from PE header. They do not give a complete list of used features, so we use all the features from PE header that we described in Section 5.1.2.
- Ahmadi et al.[27] proposed several categories of features, and we implemented most of them. However, we could not extract features specific for IDA output because we used a different disassembler. They also selected only 93 most frequent opcodes and 794 imported functions, but we select 1000 opcodes and 1000 libraries based on Fisher score. Another difference is that they only used a histogram of string lengths instead of strings by themselves. They used properties of just some selected libraries as features, but we take all sections into account. Our changes should improve the model because we improved the features that had the worst score in their paper.

The results are compared in Table 6.9. We compared our final model with a model using only PE headers in Section 6.1.1. Set of PE header features has FPR 0.00171 compared to 0.00061. Whereas model inspired by [27] with TPR 0.87419 shows comparable results to our model having 0.87363, but our model has slightly better FPR 0.00061 compared to 0.00076. It can be caused by changing the feature selection method. Fisher score does not take into account already selected features, and when there are redundant features with a higher score, we select them anyway. Therefore, selecting features per categories can help overcome this shortcoming. Another solution

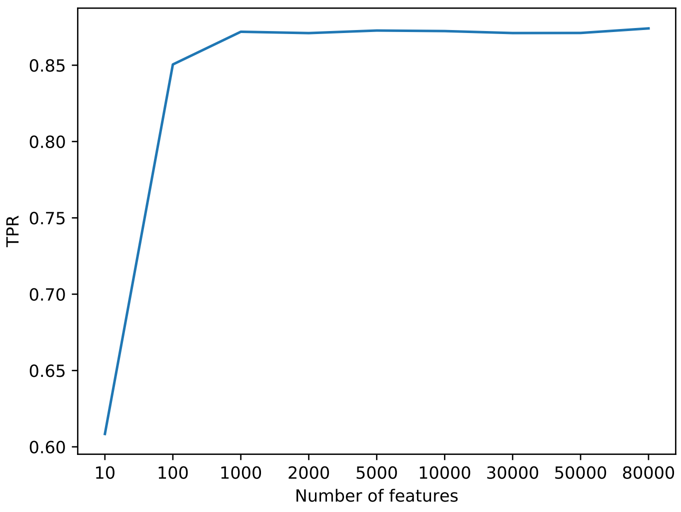


Figure 6.9: TPR dependency on number of used features.

would be selecting more features in total, because as we showed in Section 6.5 increasing number of features helps to improve the performance.

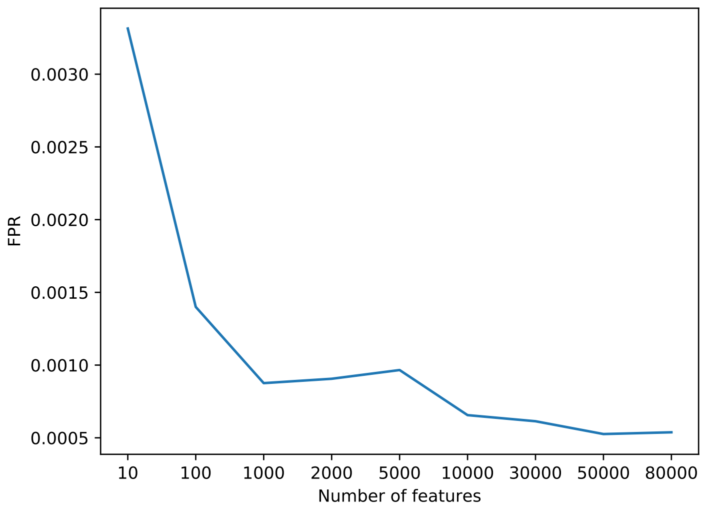


Figure 6.10: FPR dependency on number of used features.

	TPR	FPR	Remarks
Our model	0.87363	0.00061	PE header features + Raw binary features + Assembly features
Elovici et al.[9]	0.85579	0.00171	PE header features
Ahmadi et al.[27]	0.87419	0.00076	imported functions, strings, number of sections, file size, entry point, haralick, local binary patterns, entropy window, byte onegrams, size of assembly file, special symbols, registers, opcode onegrams, section properties, data define

Table 6.9: Comparison of our model with previous work.

Chapter 7

Conclusion

In this thesis, we compared features used in the static analysis for detecting malicious PE files. We extracted features from raw byte code, PE header, and assembly code. We evaluated each feature separately as well as various combinations. We found out that the best performing set of features is the one using all features we described in this thesis. We summarized feature importances and concluded that raw byte code features are rated the highest. On the contrary, assembly features did not meet our expectations. They provided only a slight improvement in the performance and showed poor performance when used by themselves. To make our analysis more concrete, we named the most important features in the model to see which n-grams, sections, etc. have the highest weight during classification.

Furthermore, we added a detailed analysis of obtained classification model. Rescaning binaries showed that our model discovered several binaries that used to be misclassified by many antivirus engines. It indicates that we are meeting the edge of malware detection where labels can be unstable and still evolving. Additionally, we examined a set of false negative samples manually and realized that we are meeting borders of static analysis because many binaries are packed or use other techniques that make static analysis almost impossible.

Besides, we tested several parameters of used LightGBM classifier as well as parameters of feature selection. We tuned the classifier to improve its detection rate further and obtained a model with TPR 0.87599 and FPR 0.00065. In case of feature selection, we examined how altering number of features influences the performance and training time of the model. We concluded that time grows linearly with the number of used features, and more features help to improve the performance up to some limit.

In future work, we suggest extracting more sophisticated features from assembly code. It contains the full functionality of the binary, so methods for mining this information should be developed. Secondly, more time should be invested in improving unpacking of binaries. Packing helps to evade detection by static analysis, and therefore robust unpacker should be proposed. Concerning particular features, strings contain valuable information about the binary, but unfortunately, they are very noisy. There is a great potential in identifying relevant strings and representing them efficiently. Finally, transforming binaries into images shows promising potential and the method should be analyzed and interpreted properly.

Appendix A

Bibliography

[1] Mahotas: Computer Vision in Python. `https://mahotas.readthedocs.io/en/latest/`. [Online; accessed 3-April-2018].

[2] OSDev Wiki. `https://wiki.osdev.org/PE`. [Online; accessed 3-April-2018].

[3] Symantec corporation. `https://www.symantec.com/security-center/writeup/2017-041417-0413-99`. [Online; accessed 24-April-2018].

[4] Virus Total. `https://www.virustotal.com/`. [Online; accessed 3-April-2018].

[5] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 183–194. ACM, 2016.

[6] AV-TEST. Av-test - independent it-security institute. `https://www.av-test.org/en/`. [Online; accessed 3-May-2018].

[7] Avast Software. RetDec: A retargetable machine-code decompiler. `https://retdec.com/`. [Online; accessed 20-May-2018].

[8] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177–186. IEEE, 2008.

[9] Y. Elovici, A. Shabtai, R. Moskovitch, G. Tahan, and C. Glezer. Applying machine learning techniques for detection of malicious code in network traffic. In *Annual Conference on Artificial Intelligence*, pages 44–50. Springer, 2007.

[10] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we need hundreds of classifiers to solve real world classification problems. *J. Mach. Learn. Res*, 15(1):3133–3181, 2014.

[11] E. Gandotra, D. Bansal, and S. Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 5(02):56, 2014.

[12] T. R. Golub, D. K. Slonim, P. Tamayo, C. Huard, M. Gaasenbeek, J. P. Mesirov, H. Coller, M. L. Loh, J. R. Downing, M. A. Caligiuri, et al. Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. *science*, 286(5439):531–537, 1999.

[13] R. M. Haralick. Statistical and structural approaches to texture. *Proceedings of the IEEE*, 67(5):786–804, 1979.

[14] T. Hastie, R. Tibshirani, and J. Friedman. The elements of statistical learning. 2001. 2001.

[15] Kaggle competitor. Kaggle: Say no to overfitting. `http://blog.kaggle.com/2015/05/26/microsoft-malware-winners-interview-1st-place-no-to-overfitting/`. [Online; accessed 5-May-2018].

[16] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3149–3157, 2017.

[17] J. Z. Kolter and M. A. Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478. ACM, 2004.

[18] Machine Learning Explained. Lightgbm and xgboost explained. `http://mlexplained.com/2018/01/05/lightgbm-and-xgboost-explained/`. [Online; accessed 21-May-2018].

[19] M. Milkovic. Generic unpacker of executable files. In *Excel@FIT*, 2015.

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

[21] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, page 4. ACM, 2011.

[22] T. Ojala, M. Pietikainen, and T. Maenpaa. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *IEEE Transactions on pattern analysis and machine intelligence*, 24(7):971–987, 2002.

[23] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, 42(3):145–175, 2001.

[24] G. Piatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. *ACM Sigmod Record*, 14(2):256–276, 1984.

[25] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. Malware detection by eating a whole exe. *arXiv preprint arXiv:1710.09435*, 2017.

[26] Reversing Labs. Titanium cloud file reputation service. `https://www.reversinglabs.com/products/file-reputation-service.html`. [Online; accessed 3-April-2018].

[27] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi. Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135*, 2018.

[28] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001.

[29] A. Shabtai, R. Moskovitch, Y. Elovici, and C. Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *information security technical report*, 14(1):16–29, 2009.

[30] The Guardian. Massive ransomware cyber-attack hits nearly 100 countries around the world. `https://www.theguardian.com/technology/2017/may/12/global-cyber-attack-ransomware-nsa-uk-nhs`. [Online; accessed 20-May-2018].

[31] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM, 2009.

[32] R. Zak, E. Raff, and C. Nicholas. What can n-grams learn for malware detection? In *Malicious and Unwanted Software (MALWARE), 2017 12th International Conference on*, pages 109–118. IEEE, 2017.