SAPIENZA
UNIVERSITÀ DI ROMA

Investigation of static features for improved APT malware identification

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea Magistrale in CyberSecurity

Candidate

Leonardo Sagratella
ID number 1645347

Thesis Advisor

Prof. Riccardo Lazzeretti

Co-Advisor

Dr. Giuseppe Laurenza

Academic Year 2019/2020

Thesis not yet defended

Investigation of static features for improved APT malware identification
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: leonardosagratella@gmail.com

Dedicato a

me stesso una stelle nascente e molto simpatica

ma soprattutto alla mia stella e luce, FEDERICO DI MAIO, figlio di

GIGGINO DI MAIO nostro ex-premier nonchè padre fondatore del reddito di

cittadinanza

Abstract

In the last years, a new cyber-threat is spreading fast, it is know as *Advanced Persistent Threat* (APT). An APT is a criminal or a group that aims to penetrate and exflitrate information from high-valuable target. They rely on malware and tools developed for the specific target they want to attack, usually governments, critical infrastructures or big organizations. Their goal is to break into the target's systems and stay undetected as long as possible. Meanwhile, they steal valuable information and try new ways to infect other target's computers. We propose a framework for malware prioritization to detect, among all the new samples that every day researchers find, the one that can be correlated to an APT group. Those samples are dispatched to analysts for further and manual inspection. To make this process as fast as possible we rely only on static features, without the need of executing the sample. We applied different techniques of feature selection to reduce the dataset dimensionality. We trained a *RandomForestClassifier* with the features left and we analyze its performances. In particular, we are interested in precision, we do not want the analysts to waste time on false positives. Overall, we obtained great performances on both precision and accuracy.

Contents

Chapter 1

Introduction

In the last decades, we are witnessing the constant improvement of technology and its utility in our every-day life. As technology advances, the cyber-threat arises. Recently the number of malware increased rapidly, in the last quarter of 2017, McAfee discovered 63.4 million new malware [1], the highest of all time. In 2018 and 2019 AvLab reported on average 12.05 million new samples per month [2]. Among those malware, there is a small group, known as Advanced Persistent Threat (**APT**), that is threatening and continuously increasing.

The National Institute of Standards and Technology (**NIST**) defines the **Advanced Persistent Threat** as [3] *"An adversary that possesses sophisticated levels of expertise and significant resources which allow it to create opportunities to achieve its objectives by using multiple attack vectors (e.g., cyber, physical, and deception). These objectives typically include establishing and extending footholds within the information technology infrastructure of the targeted organizations for purposes of exfiltrating information, undermining or impeding critical aspects of a mission, program, or organization; or positioning itself to carry out these objectives in the future."*

The term APT refers to an advanced adversary or a group,that aims to attack some critical infrastructure, a government, or an organization. They use unique attack vectors and ad-hoc malware developed for a specific target,

to attack systems and exfiltrate valuable information.

To understand better what is an APT, we need to decompose the word: [4]

Advanced: the people behind the attack have an advanced level of expertise, resources, and money. They usually do not use known malware, but they write their malware specific to the target they want. Moreover, they can gather information on the target from the intelligence of their country of origin.

Persistent: The adversary does not aim to gain access in the most number of system, but rather to have persistent access to the infrastructure. The more time they remain undiscovered in the organization's network infrastructure, the higher are the chances of lateral movement, the greater are the information they can gather. Persistent access is the key to every APT.

Threat: As said before, this is an organized threat, with a strategical vision of what to achieve. It is not an automatic tool that attacks everything trying to gather something. It is a meticulously planned attack that aims to obtain certain information from a given organization.

In general, APTs aim to higher-value targets like other nations or some big corporations. However, any individual can be a target. FireEye publish a report each year about the new APT campaign, the diagram below states which industry is the most attacked in the last year.

Advanced persistent threats, contrary to regular malware, are composed of different phases, each of which has an important role.

The attack is decomposed into smaller steps, for example, if a group of hackers wants to attack a CEO of a given company, they will not send directly to the CEO a phishing email, because it's likely that he has a complex system of security and they would be detected instantly.

Instead, the first step would hack a person in the same company with lower permissions that can have minor defense mechanisms, or hack some organization that works with the one we want to attack. Once they got the

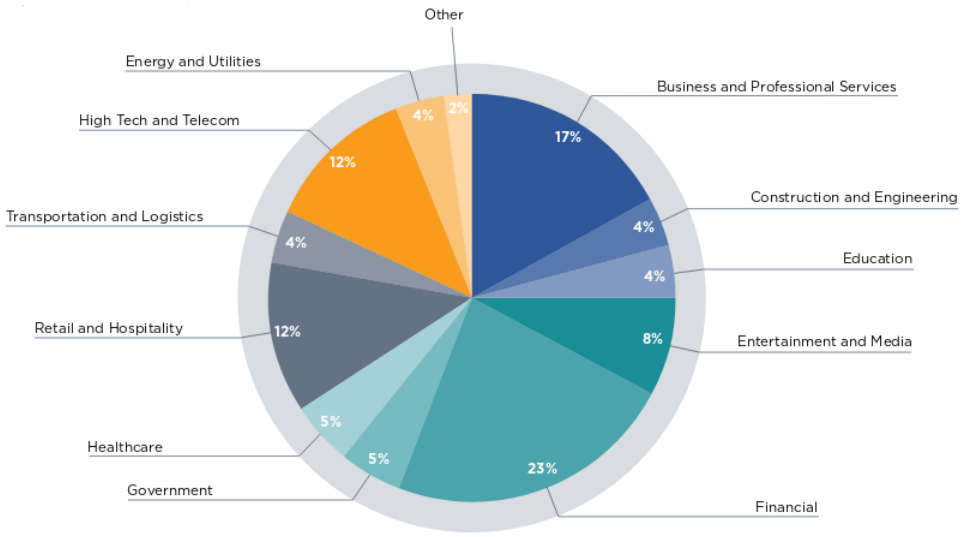


Figure 1.1. Diagram of industry target

first computer, they can explore the network infrastructure of the organization, and then decide which action is the best. They could cover their track from the log system, or locate the data they need or send a phishing email to the CEO from the owned user.

So how does an APT work? Fireeye described their behavior in six steps. [5]

1. The adversary gains access into the network infrastructure, installing a malware sent through a phishing email or by exploiting some vulnerability.

2. Once they comprised the network, the malware scans all the infrastructure looking for other entry points or weaknesses. It can communicate with a Command & Control server (C&C) to receive new instructions or to send information.

3. The malware typically establishes additional points of compromise to ensure that the attack can continue even if a position is closed.

4. Once the attackers have a reliable connection to the network, they start dumping data such as usernames and passwords, to gain credentials.

5. The malware sends the data to a server where the attackers can receive the information. Now the network is breached.

6. The malware tries to cover its tracks cleaning the log system, but the network is still compromised so the adversary can enter again if they are not detected.

Since APTs targets Critical Infrastructures, Governments, Organizations, they are mighty and critical. Analysts and researchers work every day to dissect and analyze malware. However, due to their constant increase, researchers can no more handle all of them. We proposed a prioritization system to dispatch to analysts only the samples that can be related to an APT campaign. Our system uses static features from code analysis to train a model to classify each malware.

In Chapter 2, we analyze all the paper that we use to start this thesis. We talk about a novel framework for APT malware prioritization proposed by Laurenza et al. In the next section, we analyze the work made by Caliskan et al., where they show how it is still possible to de-anonymize different programmers just relying on executable files. The last work of Dubyk shows the usage of an undocumented section of the PE header, Rich Header, in malware classification.

Chapter 2 introduces the preliminaries notions needed for the comprehension of this thesis. It focuses on Reverse Engineering (**RE**) and shows which documents are useful for an analyst in code analysis. Then we present the state-of-the-art of RE tools, why we chose Ghidra, and all its functionality useful for code analysis. Furthermore, we present the tools used for machine-learning tasks.

Chapter 4 focuses on how we extracted features from the dataset using Ghidra. We shows the different block of features proposed in Chapter 3, and for each them, we present the type of features selected and how we accomplished the task.

Chapter 5 introduces all the machine-learning models and techniques used. It presents the *RandomForest* and *XGBoost* classifier, the problem of validating results obtained from a model. Moreover, we present the different techniques for feature selection used to shrink our feature vector.

Chapter 6 comprehends all the tests we made to evaluate our models. We present in detail the results of feature selection and how we chose the best for our purpose. Furthermore, we analyzed the performances of the two models.

In Chapter 7 we show the future works we could make to improve this thesis.

Chapter 2

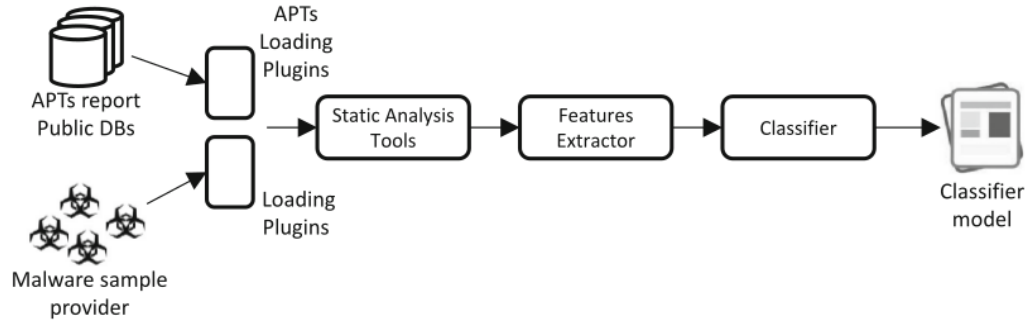
Related works

This chapter presents the related works we started from to develop this work. Firstly we introduce a paper from Laurenza et al. that explains how to build a malware triage framework for APT classification. The second work, from Caliskan et al., shows how is it possible to de-anonymize different programmers even with a compiled executable. The last work of Dubyk propose an approach for malware similarities based on the Rich Header. We expand the work of Laurenza et al. introducing new static features with the same approach of Caliskan. et al. paper. Furthermore, we use the Rich Header features as part of our dataset.

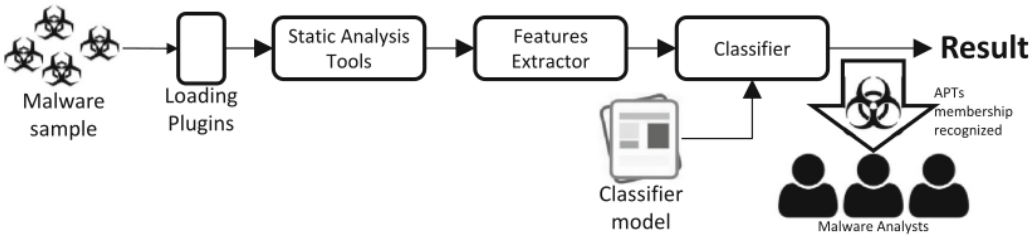
2.1 Malware Triage Based on Static Features and Public APT Reports

Laurenza et al. show that it is possible to help an analyst lightening the number of samples to analyze, using a prioritization system [6].

They propose an architecture for sample prioritization, which, based on a rank score, decides which sample has the priority to be further inspected by an analyst. In this way, it is possible to avoid wasting time analyzing samples that are not important.



(a) Training phase



(b) Testing phase

Figure 2.1. Malware triage flow.

They decide to rely only on static analysis instead of dynamic analysis to improve the speed of processing. In this architecture, the time elapsed for analyzing a classifying the samples is more important than the accuracy of the classification. Since it relies on static analysis, there is no need for sample execution, or to set up a complex virtualized environment. Every sample is processed, and some features are extracted from the header. The knowledge base created is used to classify new malware.

They propose a malware triage architecture, based on the identification of malware similar to other malware known to be related to some APTs campaign. The idea is to collect public APT reports and their related samples. Then, extracting features from the samples and assigning them to the class of the APT they belong to. Those features vectors are used to train a classifier.

Figure 2.1 shows the architecture of both the training and testing phases. The APTs loading plugin frequently loads the information on APT crawled

from public reports on the internet. The loading plugin feeds the system with new samples from different sources. The features extraction phase produces the features vectors from the malware, continuously updated by the previous loading plugin. In the classifier phase, a classifier is trained with features vector.

In the analysis phase, novel samples follow the same pipeline; the features extraction stage extracts the features vector that is passed to the classifier. If the output of the classifier is positive, i.e., the sample is related a know APT, then the sample follows a different path, and is sent a human analyst for further inspection, otherwise it is discarded.

Unfortunately, this process has some drawbacks. First of all, it is possible to identify only samples correlated to a known APT campaign, if the sample belongs to a new never investigated APT, then it is impossible to detect it. Furthermore, even if the executable belongs to a known APT, there is no guarantee that the classifier detects it because it just relies on information present in the header of the file. The malware writer can hijack that information to mislead the model.

Instead, deriving the features from the code analysis, and not only from the header, can mitigate this problem. However, it could still be possible to mislead the classifier modifying the code itself or using some code from other known APTs.

Sadly with our approach of using code analysis, the main problem is the dimensionality of the dataset. As we experienced, the number of features extracted can be huge, so feature selection is fundamental.

2.1.1 dAPTaset

Laurenza et al. have released **dAPTaset**, a public database that collects data related to APTs from existing public sources through a semi-automatic methodology and produces an exhaustive dataset [7].

The database contains information about APT campaigns and their relation with public reports and samples. In our work, we are not interested in all the information contained in this database, but just a few of them. In particular, we need the relation between md5 of the file and APT campaign name. We extracted those columns from the database and saved in a `.csv` file for future use during features extraction.

The databes contains more than 2000 samples belonging to 15 different APTs: *APT28, APT29, APT30, Carbanak, Desert Falcon, Hurricane Panda, Lazarus Group, Mirage, Patchwork, Sandwork, Shiqiang, Transparent Tribe, Violin Panda, Volatile Cedar, Winnti Group.*

Unfortunately, the dataset is not big enough and is not perfectly balanced, due to the nature of APT. It contains only 2086 samples because there are not many malware belonging to an APT campaign. Instead, the majority of public analyzed executable are just malware.

2.2 De-anonymizing Programmers from Executable Binaries

In this paper, Caliskan et al. presented their approach to de-anonymize different programmers from their compiled programs [8]. They used a dataset of executables from Google Code Jam, and they show that even after compilation the author fingerprints persist in the code, and it is still possible to de-anonymize them.

Their paper is an entry point for our work, and we tried to apply the same approach to the APT triage problem. Their approach was to extract distinct blocks of features with different tools and then analyze them to determine the best ones to describe the stylistic fingerprint of the authors precisely. Firstly, with a disassembler is possible to disassemble the binary and to obtain the low-level features in *assembly* code.

Then with a decompiler, they extracted the *Control Flow Graph* and the *Abstract Syntax Tree*. Using the *CFG*, the *AST*, and the *disassembled code* they derived the features set.

For this purpose, they used **ndisasm** for disassembling the binaries. Then using **radare2**, they generated the *Control Flow Graph*. They used **Hexrey decompiler** for decompiling the executables and generating their source code. Lastly they used **Joern**, a C fuzzy parser, to parse the output of **Hexrey decompiler** and to produce the *Abstract Syntax Tree*.

They used different types of feature selection techniques, such as WEKA's information gain, to reduce the number of features to only 53. They trained a *RandomForestClassifier* with the dataset created to de-anonymize the authors correctly.

However, the tools used by Caliskan et al. are outdated and no more maintained, so we decided to use the novel open-source tool Ghidra to write the script and extract the information we want. In this way, we significantly reduced the amount of time for feature extraction.

2.3 Rich Header

Maksim Dubyk [9] shows how the rich header produces a robust series of data points that can be exploited in static PE-based malware detection. The Rich Header is an undocumented section of *Portable Executable* (**PE**) header and provides a view of the environment where the executable was built. Firstly, Dubik shows the location of the Rich Header, how to understand and decrypt it, and how to extract it. Then, he shows different techniques to leverage the header for PE-based malware classification. The Rich Header is part of our features vector.

When building a PE, there are two distinct phases. The first one is the *compilation phase*, where the high-level instructions from programming

language are compiled into machine code, executable by the computer. The second one is the *linking phase*, a combination of those different machine code objects into a single executable. The act of combining different objects creates the Rich Header. However, only executables compiled with Microsoft Linker contain the rich header. Other compilers, such as gcc or .Net, do not have it.

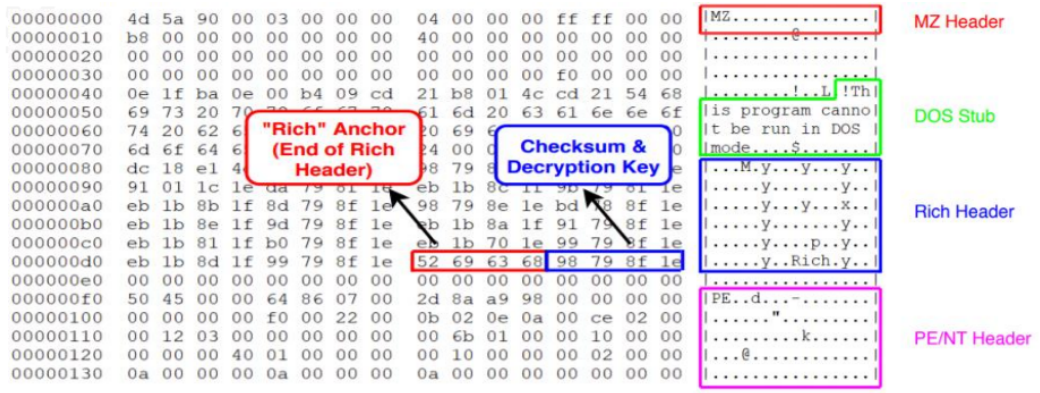


Figure 2.2. Hexview of cmd.exe

Analyzing an executable with a hexdump, the Rich Header starts from location `0x80` until the bytes sequence `0x52696368`, which means **"Rich"** in ASCII. Figure 2.2 shows the location of Rich header and the ending string "Rich". The content of the rich header is encrypted, and the corresponding decryption key and checksum are located right after the "Rich" string.

The algorithm that computes the decryption key takes advantage of two checksum algorithms. The first checksum is generated from the bytes that make the DOS header, but with the `e_lfanew` field zeroed out[10]. The second checksum is the combination of each value in the array generated. The outputs of the checksum algorithms are summed together and then masked with the bytes `0xFFFFFFFF`, the key generated is then used to encrypt the rich header section using the XOR.

It is straightforward to decrypt the rich header section. The key found after the "Rich" string is XORed backward with the chunks of the section until the end string "DanS" is found.

The decrypted rich header is an array that stores metadata related to each phase of the linking process. Each array's element is an 8-byte structure that contains three fields: *product identification* (**pID**), *product version* (**pV**), and a *product count* (**pC**). Those numbers identify the Microsoft product used in that linking phase. However, since the Rich Header is an undocumented section, there is no official mapping of pID with Microsoft products. Nevertheless, some researches partially mapped out some pID with known compilers [11].

To calculate the rich header of each sample, we used the python script provided in [9].

Chapter 3

Preliminaries

forse da ampliare disassembler e decompiler

This chapter focuses on the information needed to understand this thesis and the choice we made. We firstly introduce reverse engineering, with all its components. Then we present the state-of-the-art in reverse engineering tools, the reasons we choose Ghidra, and its features. Lastly, we discuss scikit-learn and Jupyter notebook, both used in machine learning tasks.

3.1 Reverse Engineering

Reverse engineering is the process of decomposing a human-made object to understand the underlying architecture, how it works, or to extract some information from it. This process can be applied in various fields, such as computer science, electronic, mechanical, or chemical [12].

We focus on reverse engineering applied to computer science.

The Institute of Electrical and Electronics Engineers (**IEEE**) states that reverse engineering is *"The process of analyzing a subject system to identify the system's components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction."*, where the

"subject system" is referred to the software development [13].

When somebody writes a software, he writes it with a language that is understandable by a human, for example, C or Java. But the computer cannot read it, so the programmer needs to compile the source code to let the computer understand what the software should do.

The compiling process is the process of translating the source code into a language understandable by a computer. But once the program is compiled, a human can no more read it, unless it has the corresponding source code.

If we want to understand a binary executable, but its source code is not available, we need to reverse it. There are different techniques of reversing for binary executable: disassembling the binary using a disassembler, decompiling the binary with a decompiler, or analyzing the information exchanged with a bus sniffer or a packet sniffer.

3.1.1 Disassembly code

Assembly language is a low-level programming language that has a substantial correspondence with the architecture's machine code. The program used to convert the assembly instruction to machine code instruction is called assembler. Since the assembly depends on the machine code, there is an assembly language, with its assembler, for each architecture.

With a disassembler, it is possible to revert the actions made by the assembler, so it's possible to translate machine code instruction to assembly language. The output code is formatted for human readability.

3.1.2 Decompiler?

The decompiler is a program that takes as input a binary executable and produces as output a high-level representation of the source code of the program. It is the opposite of a compiler, which, given a source code, generates

an executable. The output of the decompiler can be recompiled, and the executable will have the same behavior as the first one.

Unfortunately, the decompiler is not able to revert correctly the executable, and often it produces obfuscated code. The obfuscated code behaves in the same way, but it is harder for an analyst to understand it.

3.1.3 Control Flow Graph

The Control Flow Graph is a representation, in graph format, of the execution flow of a program or application. Frances E. Allen proposed it in [14].

The Control Flow Graph (**CFG**) is a directed graph, and it is process-oriented. Each node represents a basic block, a sequence of instructions that are executed consecutively without any jump. The edges of the graph represent the path of the execution. The graph represents all the possible paths that the program can take.

There are two types of blocks: *entry blocks* and *exit blocks*. The *entry blocks* are the ones where the flow starts; the *exit blocks* are the ones where the flow ends. Figure 3.1 represents some examples of Control Flow Graph of different statements and loop.

The CFG is useful in code analysis, to determine if some portion of the code is inaccessible.

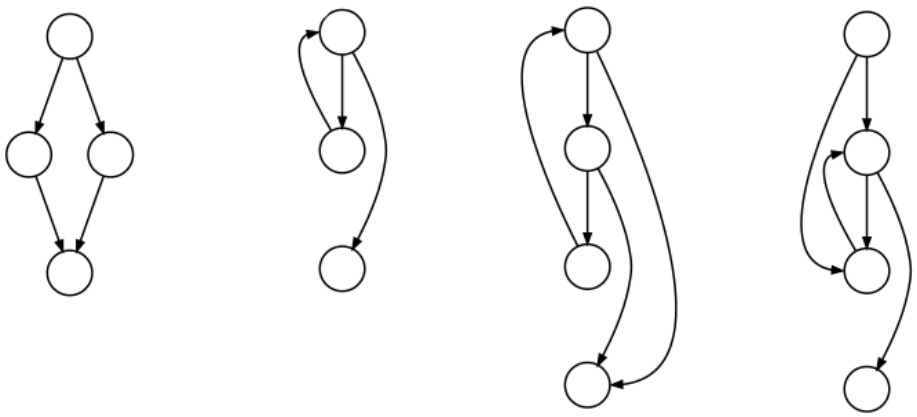
3.1.4 Cyclomatic Complexity

Cyclomatic complexity is a software metric to determine the complexity of a single function, a module, a method, some classes, or the entire program.

It is measured starting from a Control Flow Graph of the program and indicates the number of linearly independent paths of it.
The formula to calculate the complexity is:

`M = E - N + 2P`,

where `E` indicates the number of *edges*, `N` specifies the number of *nodes*, and `P`



(a) if-then-else **(b)** while loop **(c)** natural loop with two exit points **(d)** loop with two entry point

Figure 3.1. Control Flow Graph of different statements and loops

indicates the number of *connected components*.

If the entry point and exit point are connected, we have a strongly connected graph, and the formula for complexity is slightly different:

```
M = E - N + P
```

Algorithm 1 shows an example of function with different loops and statements,which corresponding Control Flow Graph is given in Figure 3.2. The numbers before the lines represent the id of the basic block to which they belong, i.e. the graph's nodes.

Being **node #1** the entry node, and **node #9** the exit code, we can manually calculate the number of independent path of the function:

- 1, 9
- 1, 2, 3, 8, 1, 9
- 1, 2, 4, 5, 7, 8, 1, 9
- 1, 2, 4, 6, 7, 8, 1, 9

Algorithm 1 Example of function with different loops and statements

```
1: while not EOF do
2:     Read record
2:     if field1 = 0 then
3:         total ← total + field1
3:         counter ← counter + 1
4:     else
4:         if field2 = 0 then
5:             Print counter, total
5:             counter ← 0
6:         else
6:             total ← total − field2
7:         end if
8:     end if
8:     Print End record
9: Print counter
```

Using the formula presented above, it is possible to calculate the complexity of the function that is equal to 4. $M = 11 - 9 + 2 \times 1 = 4$, where 11 is the *number of edges*, 9 the *number of nodes*, and 2×1 the *number of connected components*. The complexity calculated equals the number of independent paths of the graph.

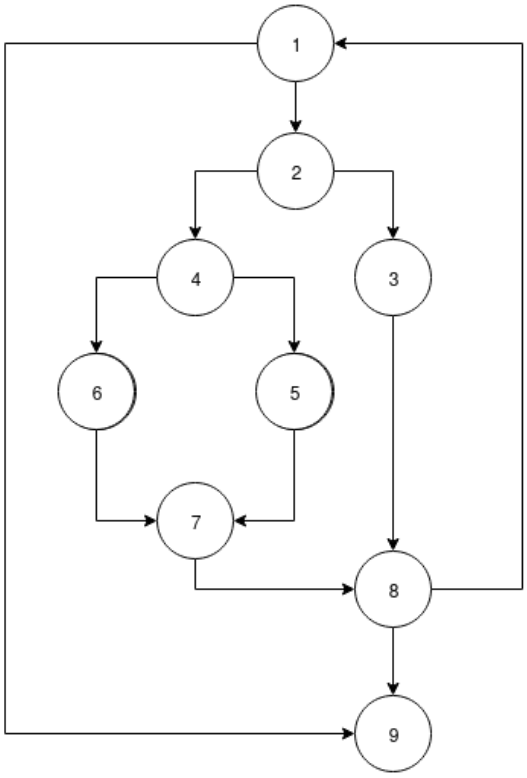


Figure 3.2. Control Flow Graph of Algorithm 1

3.2 Reverse Engineering tools

There are a lot of tools in the market that helps in reverse engineering. Some of them have tons of functionalities, and others can do just a few things.

The most important is **IDA**, a reverse engineering software developed by Hex-Rey that can achieve different things. It is available in a free version and a pro version.

It is compatible with most of the executable from different OSes, such as `PE`, `ELF`, `Mach-O`, or even raw binaries. It has extensive support and compatibility with almost every family processors.

The free version contains all the features necessary for some basic reverse engineering; it has a disassembler and performs an automatic analysis of the sample, determining the API used, which parameters are passed to them, and other information.

The analyst can navigate the code and add some notation, rename functions, and variables, to better understand the behavior of the binary. However, most of the functionalities works only with the PRO version, which costs 9000$.

The main features of IDA PRO [15] are the debugger that lets the analyst debug the executables, the possibility of writing scripts to run against the sample, and a disassembler to revert the binary into some source code.

Another famous framework for RE is **radare2** [16]. It is free and offers a decompiler and a disassembler. It is compatible with tons of processors, and executable types. It comes with a set of command-line utilities that can be used individually or together, but it also has a graphic interface to navigate the code and the possibility of running scripts. However, it is not user-friendly as other tools .

Ghidra [17] is a novel open-source reverse engineering framework developed by the National Security Agency (NSA). It facilitates the analysis of malicious code and malware like viruses and can give cybersecurity professionalists a better understanding of potential vulnerabilities in their networks and systems. It is a direct competitor of IDA PRO because it offers almost all the features that IDA PRO has, but Ghidra is entirely free. It is possible to run a headless version of Ghidra, and control it by remote. We installed it on a virtual machine on a server and use it to run headless scripts on the executables to extract information about the binaries.

We chose to run our tests on **Ghidra** for different reasons. First of all, it

is entirely free and open-source. Secondly, it has an headless version of the framework. The possibility of a headless version was perfect for our needs because we can let it run on a server, always powered on, without bothering about resource consumption. Moreover, we wanted to simplify as much as possible the extraction of features, and we do not want merging results from different tools, so we decided to use only one software. Last but not least, the scripting part of Ghidra was perfect for running analysis and extracting information from the samples.

The only inconvenient is that, since it was released less than a year ago, the documentation, the support, and the community are not well developed, so at first was hard understanding how the software works.

3.3 Ghidra

This section covers the Ghidra's features regarding the disassembled code, decompiler, Control Flow Graph and complexity presented above.

3.3.1 Disassembled code

The extraction of disassembled code is an easy and fast task. The documentation provides all the information on how to correctly use the disassembler.

Ghidra does not have a functionality to extract all the disassembled code, like in the GUI. But it is possible, for each function, to iterate the instructions in a given address space range. The instruction object has a `toString()`

method that returns the disassembled line, that we use to create the features needed.

3.3.2 PCode

PCode is a register transfer language developed by NSA for the reverse engineering framework Ghidra. The idea behind PCode is to create a language as general as possible, to let it adapt to as many different processors architecture. An intermediate language that can model the behavior of different processors is fundamental to develop a comprehensive reverse engineering framework.

The idea behind PCode is that each processor instruction can be expressed as a sequence of PCode. Developers translated each instruction into a sequence of PCode operation as input and output variables (**varnodes**).

The entire set of single PCode instruction comprises a set of arithmetic and logic instructions that almost all processors perform. The direct translation of the instructions into those operations is called raw-PCode, which can be used to emulate a single processor instruction directly. NSA developed the PCodes to facilitate the construction and the analysis of a data flow graph of disassembled instructions.

A PCode operation is the analog of a machine instruction. All PCode operations have the same format; they have one or mode varnode as input, and optionally they can have an output. Only the output varnode can be modified.

Address Space

The address space for p-code is a generalization of RAM. It represents an indexed sequence of bytes in memory that PCode can read and write into it. The address space has an identifying name, a size indicating the number of distinct indexes of memory, and an endianness that specify the encoding used to store integers and multi-byte values into the space address.

A regular processor has a RAM space to model memory accessible via its

data bus, a register space to model the processor's registers, and usually a constant space to store all the constants used by PCodes. All the data that PCode handles must be stored into an address space. PCode generally uses a temporary address space to store intermediate values when modeling the processor's instructions. The implementation of a processor can have as many address spaces as it needs.

Varnode

A varnode is a generalization of a register or memory location and has the functionality of handle the data manipulated by PCodes. It is composed of an address space, an offset into the address space, and a size. A varnode is a sequence of contiguous bytes that can be treated as a single value. The address and the size identify the varnode. Even if they have no type, some PCode operations can cast them into three types: integer, boolean, or floating-point.

In the case of integer values, the PCode operation represents the varnode using the endianness linked with the address space. In the case of floating-point, the operation uses the encoding of the varnodes that depends on its size. In the case of boolean values, the varnode has a single byte that can be 0 for false, 1 for true.

If the varnode's address space is in the constant space, the varnode is a constant or an immediate value. In this case the size of the varnode is the size or the precision available for the encoding of the constant.

3.3.3 Control Flow Graph

We rely on Ghidra's PCode representation to build our dataset for *Control Flow Graph*. Ghidra contains three different scripts for analyzing the flow of a program, and we studied those scripts to understand how Ghidra manages the PCode and their flow. The script iterates all the functions of the given sample and generates a .json file with the extracted data.

Ghidra offers a `DecompileInterface`, a class that can decompile a function, and that returns an `DecompiledResult` object with all the information needed. It is also possible to set different options to the `DecompileInterface` using the `DecompileOption` class. The resulting object contains an instance of `HighFunction`, a high-level abstraction associated with a low-level function made up of assembly instructions. The `HighFunction` object offers the possibility to iterate over the BasicBlocks of the corresponding function so we can analyze all the blocks and create our graph.

We save the information gathered into a `.json` file with the following structure: The `.json` is composed of an array of basic blocks, each of which has an index, a list of PCodes, and two lists, one containing the indexes of the previous basic blocks and the other one the indexes where the basic block points, i.e., the flow of our function. The PCodes have a field with the associated PCode operation, a list of input varnodes, and a possible output varnode.

The main problem encountered running the script is the decompilation time. Some functions are intricate, and when they come to decompile, Ghidra can take a very long time, even 25 minutes per sample. Furthermore, the `DecompileOption` has a field indicating the maximum dimension of the payload of the decompiled function. The default value is 50MB, but for some specific functions, it is still low, and we need to increase it to 500MB to correctly decompile all the functions.

3.3.4 Cyclomatic Complexity

Ghidra offers a class to compute the complexity of a function, `CyclomaticComplexity`. This class has a method to calculate the cyclomatic complexity of a function by decomposing it into a flow graph using a `BasicBlockModel`. During the decompilation, we calculate the complexity of each function and stores it into the .json file.

3.4 Scikit-learn

Scikit-learn [18] is a Python module integrating a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems.

It is open-source, commercially usable, and contains many modern machine learning algorithms for classification, regression, clustering, feature extraction, and optimization. For this reason, Scikit-learn is often the first tool in a Data Scientists toolkit for machine learning of incoming data sets.

Scikit-learn depends on *NumPy* and *pandas* modules.*NumPy* (Numerical Python) [19] is an open-source module for Python designed for mathematical and scientific computation on multi-dimensional arrays. It provides various functions for computations on arrays and matrices. *Pandas* [19] is an open-source module for Python too. It is similar to *NumPy*, and provides high-performance structures,called DataFrame, and different data analysis tools. A DataFrame is a 2d table similar to a spreadsheet, it has rows names and columns labels, and provides additional functionalities like plotting graph.

We choose Scikit-learn as a machine learning library because it is the most used in literature, and it has a great community and support. Moreover, it offers all the models and methods required by our work.

3.5 Jupyter Notebook

The Jupyter Notebook [20] is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

We choose Jupyter because it excellently fits our need for remote running. In the same server, where we installed Ghidra, we installed Jupyter, and we

enabled it to run remotely, with the proper precautions. This setup allows us to remotely work without any problem.

Machine learning tasks are often time and computationally expensive, so running them on a personal computer would be slower due to limited computer resources, and it would keep the workstation busy.

Chapter 4

Features Creation

When we decided which features could be the most representative for our model, we choose to use only static features. Extracting only static features usually do not require too much time, and there is no need for expensive virtualized systems. In dynamic analysis, the sample has to be executed in a controlled environment to analyze its behavior.

Since we want to boost up the work made by Laurenza et al. [6], we continue their path of using only static features, but we added novel features from code analysis. Our goal is a fast and efficient framework, that can analyze lot of samples without being resource expensive.

At first, we replicated the work done by Caliskan et al. [8], but we found that most of the used tools depend on software no more maintained. Some tools do not work as expected, and others were slow in processing files. To simplify the process of analyzing executables as much as possible, we decided to use only Ghidra for extracting features.

The following sections cover the blocks of features extracted, explaining the procedure used to accomplish this task.

4.1 Dataset file format

The choice of the format used for storing our dataset was crucial. We decided to save the feature vector of each sample into a `.csv` file. Using the md5, we found the related APT in dAPTaset. Each `.csv` file contains a column named md5, a column name apt, and `n` columns corresponding to the features of the sample. During the classification phase, the md5s are stripped out from the dataset and the apt column is the label of each sample.

To generate the dataset with the feature vectors of all the samples, we tried different approaches.

The first one was to merge all the `.csv` files into a single big `.csv`. Unfortunately, the more features we extracted, the more significant was the dimensionality of our dataset. **FIX HERE** When it comes to reading into python, pandas was very slow in both reading and processing the files.

A valid alternative to pandas is *Dask* [21], a flexible parallel computing library for analytics, that integrates with *pandas*, *numpy*, and *scikit*. However, the dask-ml package lacks some functionalities for the cross-validation and random forest model. Furthermore, It was still slow in reading bigger files, so we decided to find another solution to speed up the process.

Finally, we decided to store our dataset into a *Hierarchical Data Format* (**HDF5**) [22] designed to store and organize large amounts of data. This format comes with a cost, the files are much bigger, but we drastically improved the speed of reading and processing the dataset.

4.2 Disassembled features

The extraction of disassembled code is an easy and fast task. The Ghidra's documentation provides all the information on how to correctly use the disassembler. We wrote a script that extracts the disassembled code for each function and stores it into a `.dis` file. The script creates a folder for each

sample and stores the disassembled code inside of it.

From the disassembled code, we extract 5 kinds of features:

- Line unigrams;
- Disassemble unigrams;
- Disassemble bigrams;
- Instruction only unigrams;
- Instruction only bigrams;

First of all, we stripped out all the hexadecimal and numbers, replacing the regex `"\d+"` with the word `"number"`, and `"0[xX][0-9a-fA-F]+"` with the word `"hexadecimal"`. Stripping the numbers and hexadecimal reduces the possibility of overfitting because some numbers may be unique, and that would create a useless feature.

4.2.1 Line unigrams

The first block of features is the whole line unigram, we split the disassembled code of each function on the new-line character and then count all the occurrences of different line instructions. We stripped out all the commas because, in the beginning, we saved the dataset to `.csv` with comma as a separator. For example, the features generated from the function in Table 4.1 are shown in Table 4.2

4.2.2 Disassemble unigrams and bigrams

For this block of features, we split the entire line in instruction, eventual registers, or numbers. Firstly, we split the line on the first space, then if the second half of the string still contains data, we split for all the commas to get the single registers/numbers. The line `"mov eax, 0x12"` would be split in the

Table 4.1. Assembly code of a function

```
push ebx
mov eax, 1
cmp ebx, eax
jle 0xDEADBEEF
add eax, 1
cmp ebx, eax
jle 0xBACADDAC
mov eax, 0x400231BC
call eax
ret
```

Table 4.2. Feature vector of line unigrams

Feature	Value
`push ebx`	1
`mov eax, number`	1
`cmp ebx, eax`	2
`add ebx, number`	2
`jle hex`	2
`mov eax, hexadecimal`	1
`call eax`	1
`ret`	1

following array: `["mov", "eax", "hexadecimal"]` . As before, we counted the occurrences of every word in the file.

For the unigram files, we consider every word we would obtain after splitting the string as a feature. For the bigram files, instead, we consider the pair of words in the file as a feature.

Furthermore, we add a start token (`"<s>"`) before the first instruction of every function, and an end token (`"</s>"`) after the last instruction. We concatenate the first and second element of the bigram with the the string `"=>"`. The features generated from the same disassembled code are presented in Table 4.3 and Table 4.4.

4.2.3 Instruction only unigrams and bigrams

For the last block of features, we study only the frequency of the different instructions in the code, without considering the registry. As in the line bigrams, we added a start and an end token to avoid linking two instructions from different functions. Tables 4.5 and 4.6 show the features generated from

Table 4.3. Feature vector of disassemble unigrams

Feature	Value
push	1
ebx	3
mov	2
eax	6
number	2
cmp	2
jle	2
hex	3
add	1
call	1
ret	1

Table 4.4. Feature vector of disassemble bigrams

Feature	Value
<s>=>push	1
push=>ebx	1
ebx=>mov	1
mov=>eax	2
eax=>num	2
num=>cmp	2
cmp=>ebx	2
ebx=>eax	2
eax=>jle	2
jle=>hex	2
hex=>add	1
add=>eax	1
hex=>mov	1
eax=>hex	1
hex=>call	1
call=>hex	1
hex=>ret	1
ret=></s>	1

the previous example.

4.3 Control Flow Graph features

For *Control Flow Graph* features we proceed by creating a `.json` file with all the information gathered from the Decompiler as explained in Subsection 3.3.3.

From the CFG files, we extract three kinds of features:

Table 4.5. Feature vector of instruction only unigrams

Feature	Value
push	1
mov	2
cmp	2
jle	2
add	1
call	1
ret	1

Table 4.6. Feature vector of instruction only bigrams

Feature	Value
<s>=>push	1
push=>mov	1
mov=>cmp	1
cmp=>jle	2
jle=>add	1
add=>cmp	1
jle=>mov	1
mov=>call	1
call=>ret	1
ret=></s>	1

- Control Flow Graph unigrams complete;
- Control Flow Graph unigrams Pcode only;
- Control Flow Graph bigrams Pcode only;

4.3.1 Control Flow Graph unigrams complete

This first set of features contains the unigrams of the complete Pcode representation. For each sample we extract, from the corresponding `.json` file, all the pcodes in the function. We consider any unique combination of pcodes with input and output varnodes as a feature. The key for each feature is the concatenation of the PcodeOP, the input and output varnodes. In particular, we construct the key as follow: `PCodeOP_nodeoutput#nodeinput*count` of nodes. The key of the example in 4.1 is `call_ram#const*2`. As before, we count the occurrences of each key and we build our dataset.

Listing 4.1. Example of .json format with PCode

```
{
"pcodes": [
        {
        "code": "CALL",
        "varnode_in": ["ram","const"],
        "count": 2
        }]
}
```

4.3.2 Control Flow Graph Pcode only unigrams and bigrams

These two sets of features contain the unigrams and bigrams of the pcode only. We build the key using only the pcode operator, and then we count the occurrences. For the bigrams, we concatenate as before the key with the string `"=>"`.

4.3.3 Standard Library

One primary task of reverse engineering binary code is to identify library code. Since what the library code does is often known, it is of no interest to an analyst. Hex-Rays has developed the IDA FLIRT signatures to tackle the problem.

Function ID is Ghidra's function signature system. Unfortunately, Ghidra has very few Function ID datasets. There is only function identification for the Visual Studio supplied libraries. Ghidra's Function ID allows identifying functions based on hashing the masked function bytes automatically[23].

We exploit this functionality to determine which of the functions belongs to a standard library and add a boolean field in the `.json` file, indicating whether

that function is or not a standard known function. Then we calculate the presence of a function as boolean feature, 1 if the sample has it, 0 otherwise.

4.3.4 Cyclomatic Complexity

To calculate the Cyclomatic Complexity of a given function, we use the class from Ghidra's API `CyclomaticComplexity`. The function has a method that returns the complexity of the given function. We save the complexity of every function of each sample.

Then, for each sample, we calculate the complexity *maximum*, *mean*, and *mean* of all the functions that do not belong to a standard library, and use those as features.

4.4 Rich Header features

We used the script presented in paper [9] to calculate the rich header hash of each sample. Unfortunately, as pointed out in the paper, not every binary is compiled with the rich header; in fact, in *dAPTaset* only 1669 samples out of 2086 have it.

The script extracts the `productID`, the `productVersion`, and `productCount`. We concatenate those numbers with a dash `"-"` to create a key and set 1 if the sample contains the key in the header, 0 otherwise.

4.5 Total features

Table 4.7 shows the number of features of each type, with a total number of **217843** features.

Table 4.7. Number of features of each type

Features type	Dimensionality
Disassemble unigrams	2476
Disassemble bigrams	39697
Disassemble line unigrams	25927
Disassemble instruction unigrams	347
Disassemble instruction bigrams	8125
CFG unigrams	13536
CFG bigrams	118852
CFG code unigrams	61
CFG code bigrams	2026
CFG complexity	3
Standard library function	5577
Rich Header	1217

Chapter 5

Classification

This chapter presents the classification models used in our task, the validation techniques and the feature selection algorithm applied to our data.

5.1 Random Forest

Random forest [24] is an ensemble learning method for classification, regression, and other tasks that operates by constructing multiple decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

5.1.1 Decision tree

A decision tree [25] is a method used in different machine learning tasks. It uses a tree-like model of decisions and their possible consequences, including chance event outcomes, resource costs, and utility. It is one way to display an algorithm that only contains conditional control statements.

Graph's nodes represent a test on a different feature, every branch represents the outcome of the previous test, and each leaf represents the decision after analyzing all the features, i.e., the class label.

Unfortunately, the deeper is the graph, the higher are the chances of

overfitting the training test. Random forest is a method to average multiple decision trees, trained on different chunks of the training set, to reduce the variance of the output. However, this comes with a cost: an increase of the bias, and a decrease in results interpretability [26].

5.1.2 Bagging

Bagging, also known as **B**ootstrap **Agg**regat**ing**, is a machine learning meta-algorithm used to improve the accuracy of a model, reducing the model's variance and the likelihood of overfitting.

The noise in the training set affects the prediction of a single tree, but it does not affect multiple trees, as long as they are not correlated to each other. Training multiple decision trees on the same training set would produce trees highly correlated to each other. Instead, with the bootstrap aggregation technique, we can de-correlate the trees by training them on different parts of the training set [27].

Given a training set $X = x_1, \ldots, x_n$ and the corresponding labels $Y = y_1, \ldots, y_n$, the bagging algorithm repeats for B times the following process:

1. The algorithm selects a random sample with replacement of the training set X_b, Y_b

2. A decision tree f_b is trained on X_b,Y_b sets.

The predictions for unseen samples x' are calculated by taking the primary vote of individual decision tree f_b. The parameter B is free, and it can go from a few hundreds to several thousands, depending on the training set size. The optimal value can be found via cross-validation, or by examining the out-of-bag-error [28].

5.1.3 Bagging in Random Forest

In a random forest, the bagging algorithm slightly differs from the one presented above. The algorithm selects a random subset of features, a process also known as "features bagging".

The reason of this change is the correlation of trees in an ordinary bootstrap sample. If few features are a powerful predictor for the output, then most of the B trees select those features, causing the trees to be correlated. Ho [29] gives an analysis of how bagging and random subspace projection contribute to the accuracy of the model.

Commonly, in a classification problem with p features, the model uses $\sqrt{p}$ features at each split. This parameter depend on the problem, and it should be treated as a tuning parameter [26].

5.1.4 Features importance

Random forest ranks the features based on their importance to the model. Breiman [30] describes this technique in his paper.

The first step is to fit the model to the data. During this process, the model calculates the out-of-bag-error for each feature and records it. The model determines the importance of the i^{th} feature by permutating the value of the i^{th} feature against the training set, and then it calculates the out-of-bag-error again.

The average of the difference in out-of-bag-error before and after the permutations, normalized by the standard deviation of these differences, represents the feature's importance score. The higher is the score, the more important is the feature for the model.

However, this method does not work correctly with categorical variables. If these features have different levels, the model is more likely to bias the one with more levels. Using partial permutations and growing unbiased trees, it is

possible to reduce these problems [31].

5.2 XGBoost

XGBoost stands for e**X**treme **G**radient **Boos**ting. It is an open-source optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework.

XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solves many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples. [32]

The XGBoost model supports three different forms of gradient boosting: [33]

- **Gradient Boosting** with learning rate;
- **Stochastic Gradient Boosting** with; sub-sampling at the row, column and column per split levels;
- **Regularized Gradient Boosting** with both L1 and L2 regularization.

5.2.1 Gradient Boosting

Gradient boosting is a machine learning technique for regression and classification, which produces a prediction model in the form of an ensemble of weak prediction models.

Gradient Boosting is a modified version of the Boosting algorithm. Boosting is an ensemble method that converts weak learners into strong ones. It adds new models to compensate the shortcomings of existing models until the error can not be reduced anymore. [34]

Gradient Boosting is a boosting algorithm that uses a *gradient descent algorithm* to minimize the error when adding new models.

Suppose we want to train a model F to predict some values $y = F(x)$. At each step m we have a weak model F_m. To improve the model F_m, the gradient descent algorithm creates a new model F_{m+1} by adding to the previous model an estimator h such that $F_{m+1}(x) = F_m(x) + h(x)$. [35]

To find h the gradient descent algorithm starts from the perfect solution where $F_{m+1}(x) = F_m(x) + h(x) = y$, i.e., $h(x) = y - F_m(x)$. Consequently gradient boosting will fit h to the residual $y - F_m(x)$.

5.3 Cross-validation

Validation is a fundamental technique in machine learning because it allows us evaluating the stability of a model. It limits the problem of overfitting or underfitting, i.e. it makes sure that the model has low bias and variance.

Cross-validation is a model validation technique for assessing how the results of statistical analysis (model) will generalize to an independent data set.

The main idea is to split the dataset $\mathcal{D}$ into a train set $\mathcal{T}$ and a test set $\mathcal{R}$ where the union of this subset is the entire dataset and their intersection is an empty set [36]

$\mathcal{T} \cup \mathcal{R} = \mathcal{D}$

$\mathcal{T} \cap \mathcal{R} = \emptyset$

The model is trained on the training subset $\mathcal{T}$, and then is evaluated on the validation subset $\mathcal{R}$ that contains unseen data. This process can be repeated many times, using different partitions of the dataset, and then we can calculate the average of the results.

The goal of cross-validation is to test the effectiveness of the model in predicting new data, never seen in the training phase.

We have different kinds of cross-validation, leave-p-out cross-validation, k-fold cross-validation, holdout, etc. We are going to analyze the k-fold, the one used in our tests.

5.3.1 KFold

Algorithm 2 K-Fold cross-validation

1: **for** *k from 1 to K* **do**
2: $\mathcal{R} \leftarrow$ Partition k from $\mathcal{D}$
3: $\mathcal{T} = \mathcal{D} \backslash \mathcal{R}$
4: Train the model with $\mathcal{T}$
5: $Err_k \leftarrow$ Test the model on $\mathcal{R}$
6: $Err \leftarrow \frac{1}{K} \sum_{k=1}^{K} Err_k$

In K-Fold cross-validation the dataset $\mathcal{D}$ is randomly split in K sets of approximately equal size, such that [36] [37]

$$|\mathcal{D}_1| \approx |\mathcal{D}_2| \approx ... \approx |\mathcal{D}_k|$$

$$\bigcup_{k=1}^{K} \mathcal{D}_k = \mathcal{D}$$

$$\mathcal{D}_i \cap \mathcal{D}_j = \emptyset, \forall i, j \in \{1, .., K\}, i \neq j$$

For every k, the model is trained with all the samples, except for the one in $\mathcal{D}_k$, called first fold. After that the model is tested against the first fold set to estimate its performance. This process, described in Algorithm 2, is repeated for each $\mathcal{D}_k$, at each stage the error of the predictions is calculated. The estimation of total error of the model is the average of the error in the single execution.

There is no formal rule in the choice of k, usually it is 5 or 10. All we need to know is that as k gets larger, the difference in size between the training set and the resampled set gets smaller. The bias, the difference between the expected and the predicted value, decreases as k gets larger too.

Another fundamental aspect in resampling is the variance or uncertainty. An unbiased method can guess correctly but with the drawback of high uncertainty. Repeating the resampling many times we can observe a big difference between performances. However, this difference decreases as the number of resampling increases.

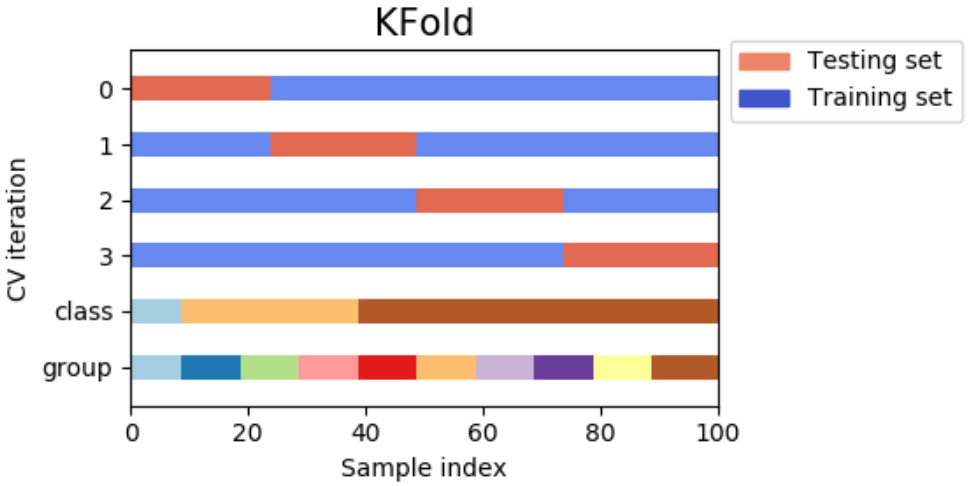


Figure 5.1. Example of 4-Fold cross-validation

The example in Figure 5.1 represents a 4-Fold cross-validation execution. For each iteration the model is trained with a different partition of the dataset, and then the average of the three execution represents the performance of the model.

Stratified KFold

When the dataset's class are not equally balanced, it is possible to have some folds without samples of a certain class. The stratification cross-validation ensures that each fold contains roughly the same proportions of classes of the entire dataset. Kohavi [38] states that stratification is normally better in terms of bias and variance, when compared to cross-validation.

Figure 5.2 depicts an execution of stratified k-fold. As illustrated in the figure, a small portion of each class is taken as testing set at each fold.

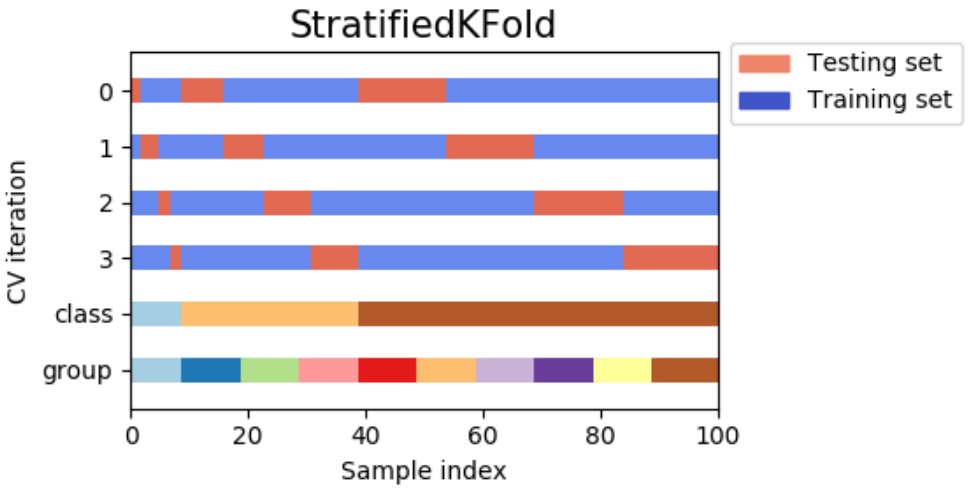


Figure 5.2. Example of stratified 4-Fold cross-validation

5.4 Feature Selection

Feature selection is a growing trend in machine learning problems. As technology advances, there is an increase in the quantity of data we can extract; this means bigger datasets to analyze and a decrease in performance and an increase in execution time.

Feature selection [39] is the process of selecting a subset of features that are more relevant for the model and ignore the rest. The main goals of feature selection are:

- Improve the performance of a classifier;
- Reduce the time and cost of analysis;
- Enhance data visualization and understanding.

The idea behind feature selection is that if the dataset contains unnecessary or redundant features, and those features can be removed without loss of information. [40]

However, there is a distinction between usefulness and relevance of a feature, a set of useful features can exclude some redundant features, that instead could be relevant to the problem [41].

Feature selection techniques can be grouped into three categories based on the approach used: **Filter methods**, **Wrapper methods**, **Embedded methods**.

5.4.1 Filter methods

Filter methods [41] are applied directly to the dataset, therefore they are independent of the model used in the prediction. Compared to methods dependent on the model, such as wrapper or embedded methods, they have a better generalization of the problem, and they are faster .

However, they have less predictive performance. They rely only on the characteristics of the features in the training set and can show the relationship between variables [42]. Usually, the filter methods rely on variable ranking. Ranking functions assign a score to each variable, and then the analyst can set a threshold of features to keep. Figure 5.3 shows the steps of filter methods.

Hastie et al. [26] state that filter methods may be preferable at first to other feature selection techniques because they are computationally and statistically scalable. It is computationally efficient because we only need to apply a function to n features in the dataset and statistically because they introduce bias, but they could have substantially less variance.

Scikit provides different functions for filter feature selection, such as mutual information, chi-square, f-classification, variance threshold. Those methods are explained in detail in the next chapter.

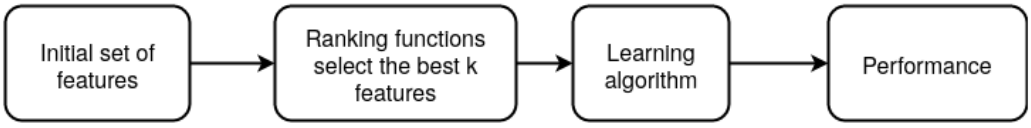


Figure 5.3. Filter method flow

5.4.2 Wrapper methods

Wrapper methods depend on the predictive model to choose a subset of features. The wrapper algorithm does not know the machine learning model used, and it is considered a black box [41].

The algorithm relies on the model to evaluate the performance of each subset of features. Each subset is used to train a model, and the model error rate establishes the score of the given subset. This procedure is repeated until an optimal subset of features is found. Figure 5.4 depicts wrapper methods execution flow.

The main drawbacks of this method are that it is very computationally expensive (Amaldi et al. [43] state that it is an NP-hard problem) and it can lead to overfitting if there are not enough data. Nevertheless, it usually gives the best result in predictive performance for the given model.

Efficient search algorithms are crucial to reducing the computational cost and time, and they are not always a synonym of decreasing in predictive performance. Greedy search algorithms are optimal for wrapper methods, and they are divided into two main categories: *forward selection* and *backward elimination* [44].

In *forward selection*, the algorithm starts from an empty set, and at each iteration, it adds a new feature to the dataset. Instead, in *backward elimination*, the algorithm starts from the whole dataset, and removes a feature at each iteration, until it finds the best subset.

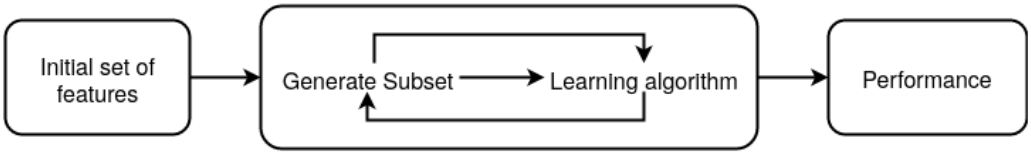


Figure 5.4. Wrapper method flow

5.4.3 Embedded methods

Embedded methods are a combination of the other two. They perform feature selection during the training process [39]. Figure 5.5 shows the flow of embedded methods. This technique allows the algorithms to be more efficient than wrapper methods. First of all, they do not need to split the dataset into training and testing sets. Secondly, they are faster because they do avoid to retrain the model for every subset of features. The most common methods are *Lasso* and *Ridge regression* and *decision tree*.

Lasso and *Ridge regression* penalize the beta coefficient by a factor, to avoid that the model focuses on a particular set of features.

Decision tree algorithms select a feature at each recursive step, during the tree growing process, dividing the samples into smaller subsets. The more child node a tree has of the same class, the more important are the features.

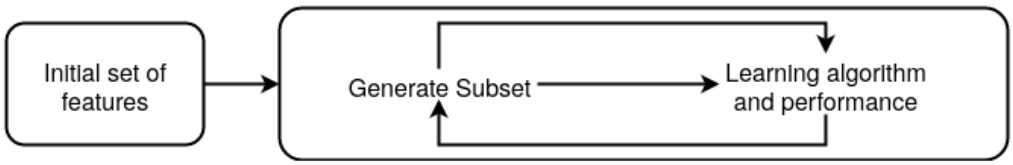


Figure 5.5. Embedded method flow

Chapter 6

Discussion

This chapter shows the results obtained during the evaluation tests. We show the feature selection technique used, and we compare the results to reach the optimal number of features in terms of time and performance metrics.

6.1 Performance Metrics

To evaluate the performance of a model, we analyze its metrics such as *accuracy, precision, recall, and f1-score.* Before explaining them we need to introduce different terms:

The *confusion matrix* (CM) is a summary of prediction results, divided by class. The rows represent the instances of actual value per class, the columns represent the instances of predicted value per class. In Table 6.1 we present an example of confusion matrix. The first row represents the instances where the actual class is "yes", in particular its intersection with the first column represents the number of samples that are correctly predicted as "yes" (value 5 in the CM). The intersection with the second column indicates the number samples that belong to the *yes* but that the classifier predicted as *no* (value 2 in the CM). The second row contains the samples that has a *no* class. The intersection with the first column indicates the number of samples predicted

as *yes* while they are a *no* (value 1 in the CM) , and the intersection with the second columns represents the number of *no* samples predicted correctly (value 4 in the CM).

Table 6.1. Confusion matrix example

	Predicted class		
Actual class		class = yes	class = no
	class = yes	5	2
	class = no	1	4

- **True Positive (TP)**: These are the observation that are positive and correct, when the value of the class is *yes* and the value predicted is also *yes* (5);
- **True Negative (TN)**: These are the observations that are negative and correct, when the value of the class id *no* and the value predicted is also *no* (4);
- **False Positive (FP)**: These are the observations that are wrong, when the value of the class is *no* and the predicted value is *yes* (1);
- **False Negative (FN)**: These are the observations that are wrong, when the value of the class is *yes* and the predicted value is *no* (2).

We can use those four parameters to calculate the metrics:

- **Accuracy**: It is the ratio of correct observations to the total observations. It can be expressed as $Accuracy = (TP + TN)/(TP + TN + FP + FN)$;
- **Precision**: It is the ratio of correct positive observation to the total of positive observation. It can be expressed as $Precision = TP/(TP + FP)$;
- **Recall**: It is the ratio of correctly predicted positive observations to the all observations in actual class. It cas be expressed as $Recall = TP/(TP + FN)$;

- **F1-Score**: It is the weighted average of Precision and Recall. It can be expressed as $F1 - score = 2 * (Recall * Precision)/(Recall + Precision)$.

Those metrics indicate the performances of a model classifier. Since we are developing a malware prioritization framework, we are interested in Precision because we do not want false positives to increase the workload of the analysts.

6.2 Testing flow?

We write a function to execute different tests using different classificators and hyperparameters.

As mentioned in Section 5.3, we used the `StratifiedKFold` class from *Scikit-learn* to cross-validate our tests. This class has a method `split` that, given the train and test subsets, it returns a range of indexes to select only a portion of the dataset, as shown in Figure 5.2. At each fold, different parts of the dataset are chosen as a test subset, but summing the test subset in all the folds, we would obtain the entire dataset. Therefore we create two arrays where, at each fold, we append the predictions made by the classification model and the ones expected. At the end of the cross-validation algorithm, we use the *Scikit-learn* functions `classification_report`, `confusion_matrix`, and `accuracy_score`, with the complete arrays of predictions and expected values, to obtain the metrics needed to evaluate the model.

We chose $k = 10$ because, as shown in the literature [38], it's the best value between performances and execution time. We set the `StratifiedKFold` option `shuffle` to true, so every time the cross-validation algorithm samples different binaries at each fold, avoiding that classification model focuses only on a very good or lousy subset of the dataset. Furthermore, we repeat each test 5 times, and then we average the results obtained.

For `RandomForest` and `XGBoost` models, we found that 150 trees are the right compromise between performances and execution time.

6.3 Feature selection

In Section 5.4, we presented different methods for feature selection. In this section, we analyze and compare them to find the best subset of features to represent APT malware.

6.3.1 Scaling dataset

The first step for selecting features is to scale the dataset. It is essential because, in our case, we have features with very different scales and contain some outliers. These two aspects can decrease the predictive performance of the classification model. We chose the `MaxAbsScaler` from *scikit-learn* because it fits our needs. The scaler does not shift or center the data, and it does not destroy the sparsity of the features. For each feature, the algorithm calculates the maximum value and scales all the features in the column, such that the maximum value is equal to 1.0. All the features will be in the range of 0 and 1.0, but they maintain their variance.

6.3.2 Remove low variance

The second step is to visualize the variance of the dataset. We calculated with the std function on axis 0, and we visualize them with matplotlib. Figure 6.1 shows the variance of the entire dataset. As we can see, some features have zero variance, and this means that they are constant over all the samples. Thus they are not useful for classification.

To remove them, we use `VarianceThreshold` class that removes all low-variance features. Is it possible to set a threshold, if a feature variance is below the threshold, then it is removed. By default, the class removes all zero-variance features. This method removes **32** features from our dataset.

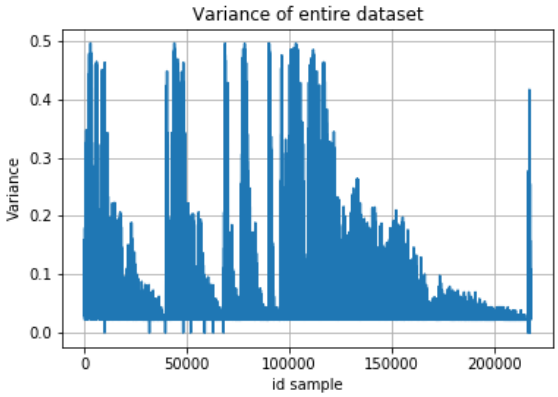


Figure 6.1. Variance of features in the entire dataset

6.3.3 Filter methods

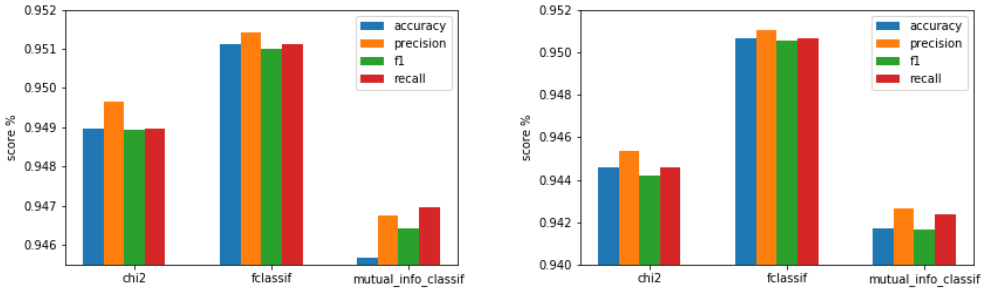
Filter methods work by selecting the best features based on a ranking function. Scikit-learn offers various scoring functions, such as *chi2, f_classif, mutual_info_classif*. To choose the best k features scikit-learn has two classes:

- `SelectKBest:` removes all but the highest k scoring features.
- `SelectPercentile` removes all but a user-specified highest scoring percentage of features.

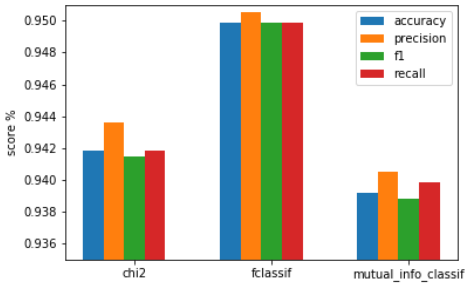
We use `SelectPercentile` to perform some tests with different percentages and compared the results. We select respectivly the 25%, 15%, and 10% of the functions `chi2, fclassif, and mutual_info` and summarized them in Figures 6.2 and Tables 6.2. The column *time* in Tables 6.2 refers to the time elapsed during the feature selection, not during the classification algorithm to test the performances.

As we can see from results, the `f_classif` is the one who performs the best, even if it takes a little more time than `chi2` . On the contrary, `mutual_info_fclassif` is the slower, it takes half a day to calculate the mutuals information of the features, and it is also the worst in terms of performance.

However, as we reduce the percentage of features, the performances degrade too. The best percentage is 25%, but it keeps over 50000 features, which are

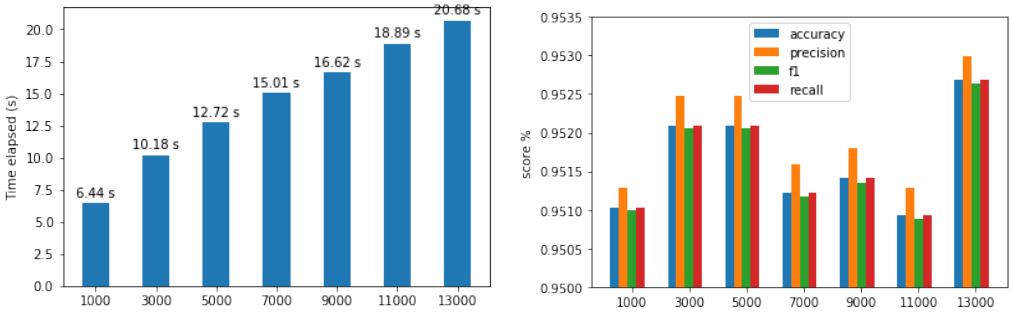


(a) Ranking functions score with 25% **(b)** Ranking functions score with 15%



(c) Ranking functions score with 10%

Figure 6.2. Comparison of metrics score for different ranking functions with different percentages selected



(a) Time comparison **(b)** Metrics comparison

Figure 6.3. Comparison of time and metrics score with different number of features selected using `SelectFromModel`

not good for our purpose. The classification time, instead, is acceptable, it varies from 58s for 25% features, to 41s for 10% features. We cannot rely only on filter methods because they would not shrink our dataset as we intended. So we tried other methods for feature selection provided by *scikit-learn*.

6.3.4 Embedded Methods

Embedded methods use models that have build-in feature selection methods. `RandomForestClassifier` has a field `feature_importances` to rank all the features from best to worst. In particular, *scikit-learn* provides a `SelectFromModel` class, that using a classifier that has a build-in feature importance method, select the best features based on the model. It is possible to set a `max_features` parameter to limit the number of selected features.

However, the `SelectFromModel` algorithm chooses 12781 features in just 18s. Even with `max_features` set to 21781 (the 10% of the entire dataset), the algorithm still chooses only the 12781 features. Comparing the performances with the filter methods presented before, we find out that `SelectFromModel` is more accurate and the fastest in classification because there are fewer features in the feature vector. Based on those tests we prefer to discard filter methods and keep using `SelectFromModel`.

Table 6.2. Comparison of metrics score for different ranking functions with different percentages

(a) Summary of ranking functions selecting the 25% of the best features

Metrics	chi2	fclassif	mutual_info
Time	7.60s	246.05s	43217.97s
Accuracy	94.90%	95.11%	94.57%
Precision	94.96%	95.14%	94.67%
Recall	94.90%	95.11%	94.70%
F1-score	94.89%	95.10%	94.64%

(b) Summary of ranking functions selecting the 15% of the best features

Metrics	chi2	fclassif	mutual_info
Time	7.60s	246.05s	43217.97s
Accuracy	94.46%	95.06%	94.17%
Precision	94.54%	95.1%	94.27%
Recall	94.46%	95.06%	94.24%
F1-score	94.42%	95.06%	94.17%

(c) Summary of ranking functions selecting the 10% of the best features

Metrics	chi2	fclassif	mutual_info
Time	7.60s	246.05s	43217.97s
Accuracy	94.19%	94.99%	93.92%
Precision	94.36%	95.05%	94.05%
Recall	94.19%	94.99%	93.99%
F1-score	94.15%	94.98%	93.89%

We run various tests with different numbers of `max features` to compare the results. The tests are shown in Figure 6.3. Figure 6.3a shows how time varies related to the number of features selected.

As we can see from Figure 6.3b there is no big difference in the tests made between *accuracy, precision,recall, or f1-score*. For example, the maximum difference in accuracy is just 0.0016s. This happens because `RandomForestClassifier` already performs feature selection using `feature_importances` during the training phase. Since the time is dependent on the number of features, we choose to balance the performances and the execution time selecting the best 3000 features.

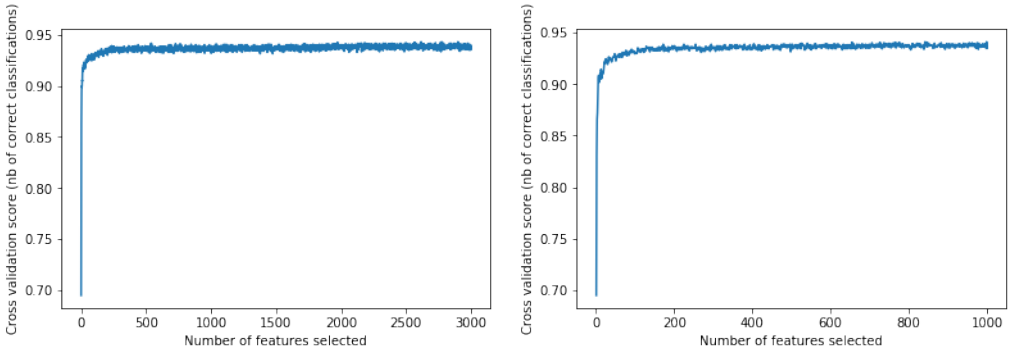
6.3.5 Wrapped methods

As explained in Section 5.4.2 wrapper methods are the most computational expensive. *Scikit-learn* has `RFE` and `RFECV` classes for recursive feature elimination.

From *scikit-learn* documentation [45] "*The goal of recursive feature elimination (RFE) is to select features by recursively considering smaller and smaller sets of features. First, the estimator is trained on the initial set of features and the importance of each feature is obtained either through a coef_ attribute or through a feature_importances_ attribute. Then, the least important features are pruned from current set of features. That procedure is recursively repeated on the pruned set until the desired number of features to select is eventually reached*".

The `RFECV` works the same as `RFE`, but it uses a cross-validation algorithm to validate the evaluation performance. The model used is `RandomForest` and `StratifiedKFold` as cross-validation technique.

Unfortunately, it is really slow. We tried using the entire dataset, but even after three days of computing, the algorithm didn't stop. We changed strategy, and we tried to use the features calculated from filter methods, but again it



(a) Initial dataset contains 3000 features **(b)** Initial dataset contains 1000 features

Figure 6.4. Relation between cross validation scores and number of features.

took more than three days. In the end, we decide to use the features selected by `SelctFromModel`. However, `RFECV` takes a considerable amount of time compared to other feature selection techniques.

In particular, we run tests using both the 3000 and the 1000 features selected before. As expected, the time increases exponentially with the number of features. The 3000 dataset took almost 1 hour to reduce to the optimal number of features, while the 1000 one took only 12 minutes. Figure 6.4 shows the decrease in performance as we remove more and more features from the dataset. Figure 6.5 shows the comparison of evaluation metrics between the two different datasets. Comparing the results, we did not notice a big difference in terms of accuracy precision or recall. Even if one algorithm stops at **2616** features and the other to only **842**. However, the calculation time was way longer for the first dataset.

We continue our tests using the output of `RFECV` for 1000 features, because it was faster to calculate. We apply again the `SelectFromModel` algorithm to lastly reduce the number of features, leaving us with only 305 features.

We stopped at 305 features, because we did not notice any improvement in terms of classification time. We consider 5.5s an acceptable time for classification. Furthermore, the less features we got, the worst are the performance in evaluation, so we decide to keep that number of features.

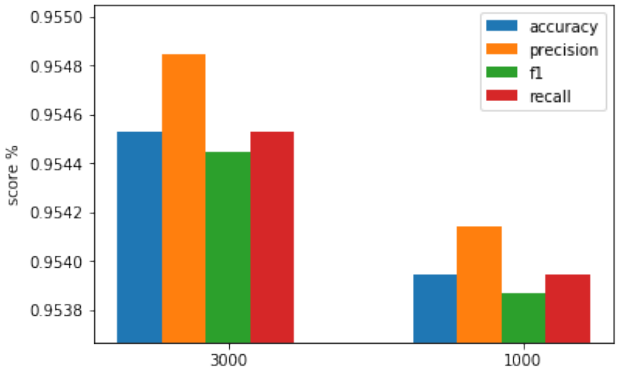


Figure 6.5. RFECV applied respectively to a dataset of 3000, and 1000 features.

Analyzing the type of features we have, we find that almost all the blocks of features still has some feature in the dataset. The Table 6.3 reports the number of features keep per type. The standard library features are removed during feature selection, the others are significantly reduced.

6.4 Model Evaluation

The following section presents the tests made on the final dataset using two different models: `RandomForestClassifier` ans `XGBoostClassifier`.

We compared the results obtained with this different models, and compare them to performances of the initial dataset.

Table 6.4 summarize the executions. In the table the term `RF` means RandomForestClassifier, `XBG` means XGBoostClassifier, and `FS` means Feature Selection. To run the final tests, we chose to set to 20 the number of repetition presented in Section 6.2 to get a more exhaustive result.

The `RandomForestClassifier` is the fastest and the most accurate. With only 5s per execution, it reaches a surprisingly accuracy of 95.21% versus the 94.76% of `XGBoostClassifier`. Compared with the original dataset we went from more than 200 thousands features to just 304. Analyzing the performances of entire dataset and the one with feature selected, we notice that the execution time is obviously slower with the entire dataset.However,the performances

Table 6.3. Comparison between initial number of features, and number of features after feature selection.

Features type	Number	Initial number
Disassemble Unigrams	19	2476
Disassemble Bigrams	62	39697
Disassemble Line Unigrams	28	25927
Disassemble Instruction Unigrams	9	347
Disassemble Instruction Bigrams	19	8125
CFG Unigrams	26	13536
CFG Bigrams	316	118852
CFG Code Unigrams	14	61
CFG Code Bigrams	94	2026
CFG Complexity	2	3
Standard Library Function	0	5577
Rich Header	4	1217

metrics increased a little with our subset of relevant features.

Table 6.5 shows a detailed representation of Precision, Recall, and F1-score divided per class. We managed to reach 100% of precision in some classes, such as *Sandwork* and *Volatile Cedar*. *Violin Panda* has the worst performance with metrics around 82%, probably because there are few samples available, only 22. Table 6.6 shows the Confusion Matrix of the execution. The labels used in the table `A,B,..,P,Q` corresponds to the APT classes presented in Table 6.5.

Table 6.4. Comparison of execution time and metrics of different models

Algorithm	**Time**	**Accuracy**	**Precision**	**Recall**	**F1-Score**
RF with FS	5.22s	95.21%	95.24%	95.21%	95.20%
XGB with FS	53.14s	94.76%	94.78%	94.76%	94.74%
RF with full dataset	366.65s	94.98%	95.03%	94.98%	94.97%

Table 6.5. Detailed values divided per APT class of Precision, Recall, F1-score, and number of samples.

APT	Precision	Recall	F1-Score	Support
APT28	89.65%	92.21%	90.9%	68
APT29	95.25%	94.32%	94.78%	205
APT30	99.06%	98.27%	98.66%	101
Carbanak	93.17%	89.61%	91.35%	76
Desert Falcon	91.9%	87.89%	89.82%	45
Hurricane Panda	98.2%	95.95%	97.06%	315
Lazarus Group	93.67%	94.05%	93.85%	58
Mirage	89.64%	83.77%	86.6%	53
Patchwork	94.97%	96.78%	95.87%	557
Sandwork	100.0%	95.45%	97.67%	44
Shiqiang	97.07%	90.0%	93.39%	31
Transparent Tribe	95.21%	97.17%	96.18%	267
Violin Panda	82.18%	82.61%	82.37%	23
Volatile Cedar	100.0%	94.71%	97.27%	35
Winnti Group	93.97%	97.13%	95.52%	176

Table 6.6. Confusion Matrix of `RandomForestClassifier` with the reduced dataset. Labels indicating APT classes are in the same order as in Table 6.5

	A	B	C	D	E	F	G	H	I	L	M	N	O	P	Q
A	63	0	0	0	0	1	0	0	0	0	0	0	0	0	4
B	1	193	0	0	0	0	0	0	6	0	0	3	0	0	1
C	0	0	99	0	0	0	1	0	1	0	0	0	0	0	0
D	3	2	0	68	0	0	0	0	2	0	0	0	0	0	0
E	0	1	0	0	40	0	0	0	3	0	0	2	0	0	0
F	2	1	1	1	0	302	0	0	3	0	0	3	0	0	1
G	0	0	0	0	0	0	55	1	1	0	0	0	0	0	0
H	1	0	0	0	0	1	3	44	3	0	0	0	0	0	0
I	0	3	0	2	3	1	0	0	539	0	0	4	4	0	1
L	0	0	0	0	0	0	0	0	1	42	0	0	0	0	1
M	0	0	0	0	0	0	0	2	1	0	28	0	0	0	0
N	0	2	0	0	1	0	0	0	3	0	0	259	0	0	1
O	0	0	0	0	0	0	0	1	2	0	1	0	19	0	0
P	0	0	0	0	0	0	0	0	0	0	0	1	0	33	0
Q	0	0	0	2	0	1	0	1	1	0	0	0	0	0	171

Chapter 7

Conclusions and Future Works

In the last decades, the number of malware increased exponentially. Between all those malware, there is a particular type known as Advanced Persistent Threat, which is really dangerous. APT is an advanced adversary or a group whose goal is to intrude secretly into computer systems of big companies or foreign governments to steal valuable information.

We proposed a framework to detect if a suspected malware belongs or not to an APT. The analysts can then concentrate their resources to investigate samples tagged as related to an APT with a certain degree of confidence. This thesis enriches the static features used in malware triage in [6], adding features from code analysis. A classifier is trained with features from disassembled code, control flow graph, cyclomatic complexity, and rich header. Even if static features are not sufficient to classify different malware, they are fast and easy to compute. For an early identification in a malware APT prioritization framework, we have promising results. The precision, i.e., the number of false positives, is crucial for this framework. We don't want an analyst to focus on a sample targeted as potential APT when instead is just a random malware. In our thesis, APT samples are identified on average with precision and accuracy over 95%, some samples, even with a precision ad accuracy of 100%. Overall, using feature selection, we reduced the classification time, and improved a

little bit the performance, with respect to the original dataset.

However, the dAPTaset is not sufficiently large, and we tested only samples known to be related to some APT. As future works, first of all, we want to test our classification model on a larger dataset. Furthermore, we want to extend this thesis also to non-APT malware. The framework should recognize which samples are non-APT and classify the APT membership of the others.

Secondly, we want to investigate further some samples that we found as packed. We are interested in writing a script with Ghidra to automatically unpack the packed malware and extract more features to enrich our dataset. Besides, we want to study more Ghidra to take advantage of all its features and develop scripts to extract more information from the binaries automatically.

Moreover, we want to test different classification models and feature selection techniques. We want to focus on *Principle Component Analysis* to diminish the dimensionality of our dataset. We want to consider deep neural networks as a classification method since many researchers reported successfully results [46] [47].

Bibliography

[1] "McAfee Labs Threats Report March 2018." https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-mar-2018.pdf.

[2] "Malware Report 2020." https://www.av-test.org/en/statistics/malware/.

[3] "NIST Glossary Advanced Persistent Threat." https://csrc.nist.gov/glossary/term/advanced-persistent-threat.

[4] ItGovernance, "Advanced Persistent Threats." https://www.itgovernance.co.uk/advanced-persistent-threats-apt.

[5] FireEye, "Anatomy of Advanced Persistent Threats." https://www.fireeye.com/current-threats/anatomy-of-a-cyber-attack.html.

[6] G. Laurenza, L. Aniello, R. Lazzeretti, and R. Baldoni, "Malware triage based on static features and public apt reports," in *International Conference on Cyber Security Cryptography and Machine Learning*, pp. 288–305, Springer, 2017.

[7] "dAPTaset." https://github.com/GiuseppeLaurenza/dAPTaset.

[8] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code

stylometry," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pp. 255–270, 2015.

[9] M. Dubyk, "Leveraging the PE Rich Header for Static Malware Detection and Linking," 2019.

[10] J. White, "Case studies in Rich Header analysis and hunting." `http://ropgadget.com/posts/richheader_hunting.html`.

[11] Dishather, "Rich print." `https://github.com/dishather/richprint`.

[12] E. Eilam, *Reversing: secrets of reverse engineering*. John Wiley & Sons, 2011.

[13] E. J. Chikofsky and J. H. Cross, "Reverse engineering and design recovery: A taxonomy," *IEEE software*, vol. 7, no. 1, pp. 13–17, 1990.

[14] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, pp. 1–19, ACM, 1970.

[15] Hex-Rays, "IDA Pro." `https://www.hex-rays.com/products/ida/index.shtml`.

[16] "Radare2." `https://rada.re/n/radare2.html`.

[17] National Security Agency, "Ghidra." `https://www.nsa.gov/resources/everyone/ghidra/`.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[19] "NumPy and Pandas Introduction." `https://cloudxlab.com/blog/numpy-pandas-introduction/`.

[20] Jupyter, "Jupyer notebook." `https://jupyter.org/`.

[21] "Dask." `https://dask.org/`.

[22] "Hierarchical Data Format." `https://portal.hdfgroup.org/display/HDF5/HDF5`.

[23] 0x6d696368, "Ghidra FID Generation." `https://blog.threatrack.de/2019/09/20/ghidra-fid-generator/`.

[24] T. K. Ho, "Random decision forests," in *Proceedings of 3rd international conference on document analysis and recognition*, vol. 1, pp. 278–282, IEEE, 1995.

[25] B. Kamiński, M. Jakubczyk, and P. Szufel, "A framework for sensitivity analysis of decision trees," *Central European journal of operations research*, vol. 26, no. 1, pp. 135–159, 2018.

[26] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media, 2009.

[27] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[28] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning*, vol. 112. Springer, 2013.

[29] T. K. Ho, "A data complexity analysis of comparative advantages of decision forest constructors," *Pattern Analysis & Applications*, vol. 5, no. 2, pp. 102–112, 2002.

[30] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[31] H. Deng, G. Runger, and E. Tuv, "Bias of importance measures for multi-valued attributes and solutions," in *International Conference on Artificial Neural Networks*, pp. 293–300, Springer, 2011.

[32] T. Chen, "XGBoost." `https://xgboost.ai/about`.

[33] J. Brownlee, "A Gentle Introduction to XGBoost for Applied Machine Learning." `https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/`.

[34] Z.-H. Zhou, *Ensemble methods: foundations and algorithms*. Chapman and Hall/CRC, 2012.

[35] C. Li, "A Gentle Introduction to Gradient Boosting." `http://www.chengli.io/tutorials/gradient_boosting.pdf`.

[36] B. Ghojogh and M. Crowley, "The theory behind overfitting, cross validation, regularization, bagging, and boosting: Tutorial," *arXiv preprint arXiv:1905.12787*, 2019.

[37] M. Kuhn and K. Johnson, *Applied predictive modeling*, vol. 26. Springer, 2013.

[38] R. Kohavi *et al.*, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Ijcai*, vol. 14, pp. 1137–1145, Montreal, Canada, 1995.

[39] I. Guyon and A. Elisseeff, "An introduction to variable and feature selection," *Journal of machine learning research*, vol. 3, no. Mar, pp. 1157–1182, 2003.

[40] M. L. Bermingham, R. Pong-Wong, A. Spiliopoulou, C. Hayward, I. Rudan, H. Campbell, A. F. Wright, J. F. Wilson, F. Agakov, P. Navarro, *et al.*, "Application of high-dimensional feature selection: evaluation for genomic prediction in man," *Scientific reports*, vol. 5, p. 10312, 2015.

[41] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artificial intelligence*, vol. 97, no. 1-2, pp. 273–324, 1997.

[42] N. Sánchez-Maroño, A. Alonso-Betanzos, and M. Tombilla-Sanromán, "Filter methods for feature selection–a comparative study," in *International Conference on Intelligent Data Engineering and Automated Learning*, pp. 178–187, Springer, 2007.

[43] E. Amaldi and V. Kann, "On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems," *Theoretical Computer Science*, vol. 209, no. 1-2, pp. 237–260, 1998.

[44] J. Reunanen, "Overfitting in making comparisons between variable selection methods," *Journal of Machine Learning Research*, vol. 3, no. Mar, pp. 1371–1382, 2003.

[45] "Recursive Feature Elimination." `https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFE.html`.

[46] H. Zhou, "Malware detection with neural network using combined features," in *China Cyber Security Annual Conference*, pp. 96–106, Springer, 2018.

[47] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, "Malware detection with deep neural network using process behavior," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, pp. 577–582, IEEE, 2016.