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 premier nonchè padre fondatore del reddito di cittadinanza

Abstract

Questa tesi parla di me.

Contents

Chapter 1

Introduction

Chapter 2

Tools

2.1 Advanced Persistent Threat

APT stands for Advanced Persistent Threat, a kind of sophisticated attack which requires an advanced level of expertise and aims to remain persistent on the attacked infrastructure.

The term APT can refer to a persistent attack with a specific target, or it can refer to the group that organized the attack, sometimes the group is affiliated with some sovereign state.

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

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. [1]

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.

A point of particular concern is the retargeting, in the Americas, 63% of the companies attacked by an APT, are attacked again last year by the same or similar group. In the Asian and Pacific areas, this is even worse, 78% of the industries are hacked again. [2]

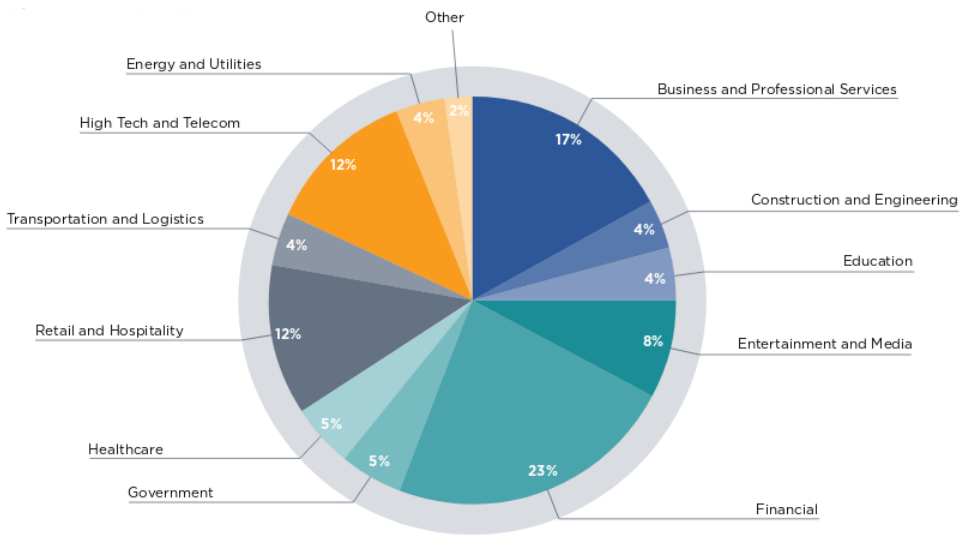
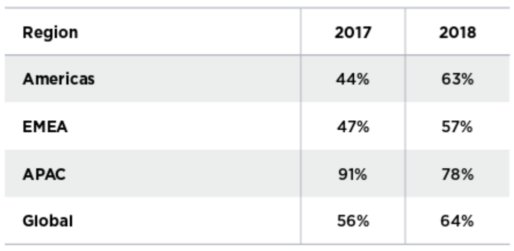


Figure 2.1. Diagram of industry target



Region	2017	2018
Americas	44%	63%
EMEA	47%	57%
APAC	91%	78%
Global	56%	64%

Figure 2.2. Retargeting divided by regions

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. Once they got the 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. [3]

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.

2.2 Ghidra

Ghidra is an open-source tool for Reverse Engineering developed and by the National Security Agency (NSA). It helps analyze malicious code and malware like viruses, and can give cybersecurity professionals a better understanding of potential vulnerabilities in their networks and systems [4]

Usually, reverse engineering is the process of analyzing something to understand how it works. In the case of a program written in Java or C or C++, the code will be readable by a human but not bt a computer. It needs to be compiled in a language understandable by the network, but once it is compiled, we can no more read it.

To understand how the program works, we need a toolkit to take it apart, and this is what Ghidra does. There are a lot of tools in the market that can do the same thing, in different ways, some of them are open-source and free, other you need to pay a license.

We choose to use Ghidra because it is free, and it offers the possibility of writing scripts to run against the binaries analyzed. In this way, we extracted all the necessary information automatically from the APT binaries.

2.3 Scikit-learn

Scikit-learn 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. [5]

Chapter 3

Related works

3.1 De-anonymizing Programmers from Executable Binaries

In this paper, Caliskan et al. presented their approach to de-anonymize different programmers from their compiled programs. 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 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**. They determine the stylistics features from those four documents.
In particular, the tools used are **ndisasm radare2** disassembler for the disassembled code and the Control Flow Graph; **Hexray** decompiler for the pseudocode, which is passed as input to **Joern**, a C fuzzy parser, to produce the **Abstract Syntax Tree**.

They used different types of features selection techniques to reduce the number of features to only 53. They trained a RandomForest Classifier with the dataset created to de-anonymize the authors correctly.

This paper is an entry point for our work, and we tried to apply the same approach to the apt triage problem. 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.

3.2 APT triage

Laurenza et al. show that it is possible to help an analyst lightening the number of samples to analyze. The main idea is to process all the executables, extract some features, and then classify them to determine if they belong or not to a possible APT campaign. The analyst can then analyze only the suspected files that can be related to some APT. Unfortunately, this work 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.

The dataset used by Laurenza et al. is **dAPTaset**, a public database that collects data related to APTs from existing public sources through a semi-automatic methodology and produces an exhaustive dataset. Unfortunately, the dataset is not big enough and is not perfectly balanced. It contains only 2086 samples because there are not many samples belonging to an APT campaign. Instead, the majority of public analyzed samples are just malware.

3.3 Rich Header

da scrivere sunto del lavoro su rich header [6]

Chapter 4

Features Creation

When we decided which features could be the most representative for our model, we choose to use only static features. We are looking for a framework fast and efficient, that can analyze lots of sample without being resource expensive.

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

Ghidra comes with a headless analyzer, which analyzes and runs scripts on the given sample. The headless version can run in any server, even without a desktop environment. So we built up a virtual machine in the Sapienza network and installed Ghidra there. Unfortunately, Ghidra's documentation is not exhaustive since it was released less than a year ago. The hardest part was understanding Ghidra's APIs and how to exploit them for our purpose. The already made scripts were useful for our task because they contain many approaches for extracting data.

4.1 Disassemble features

The extraction of disassembled code was an easy and fast task. The 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 inside the disassembled code.

From the disassembled code, we extracted 5 kinds of various features:

- Entire 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 '+' with the word 'number', and '0[xX][0-9a-fA-F]+' with the word hexadecimal. Stripping the numbers and hexadecimal reduced the possibility of overfitting because some numbers may be unique, and that would create a useless feature.

Furthermore, we create a .csv file for each sample, containing all the features calculated, the md5, used as an identifier of the executable, and the apt name. Then all the files are merged into a big .h5 with all the samples. In the first approach, we stored all the features into a .csv file, but the more features we extract, the more significant were the dimensionality of our dataset. 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, 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. In the end, we decided to store our dataset into a Hierarchical Data Format (HDF5) 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.

4.1.1 Entire line unigram

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 of the following disassembled function would be:

Table 4.1. Code for function f

```
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. Entire 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
apt	PatchWork
md5	1234dc...eb121

4.1.2 Disassemble unigrams and bigrams

For this block of features, we split the entire line in instruction, eventual registers, or numbers. We first divided on the first space, and then if the second half of the string still contains data, we split for all the commas to get the single registers/number. the line "mov eax, 0x12" would be split in the following array: ["mov", "eax", "hexadecimal"] . As before, we counted the occurrences of every word in the file.

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

Furthermore, we added a start token ("<s>")at the beginning of the function file, and an end token ("</s>") at the end of the file. We concatenate the first and second element of the bigram with the the string "=>" The features generated from the same disassembled code would be the following:

Table 4.3. 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
apt	PatchWork
md5	1234dc...eb121

Table 4.4. 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
apt	PatchWork
md5	1234dc...eb121

4.1.3 Instruction only unigrams and bigrams

For the last block of features, we decided to study only the frequency of the different instructions in the code, without considering the registry. As before in the bigrams, we added a start and an end token to avoid linking two instructions from different functions. The features from the previous example would be:

4.2 Control Flow Graph features

la parte che segue è tutta da riscrivere in quanto CTRL+C CTRL+V diretto da Ghidra doc [7]

Table 4.5. Instruction only unigrams

Feature	Value
push	1
mov	2
cmp	2
jle	2
add	1
call	1
ret	1
apt	PatchWork
md5	1234dc...eb121

Table 4.6. 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
apt	PatchWork
md5	1234dc...eb121

4.2.1 PCode

P-code is a register transfer language designed for reverse engineering applications. The language is general enough to model the behavior of many different processors. By modeling in this way, the analysis of different processors is put into a common framework, facilitating the development of retargetable analysis algorithms and applications.

Fundamentally, p-code works by translating individual processor instructions into a sequence of p-code operations that take parts of the processor state as input and output variables (varnodes). The set of unique p-code operations (distinguished by opcode) comprise a fairly tight set of the arithmetic and logical actions performed by general purpose processors. The direct translation of instructions into these operations is referred to as raw p-code. Raw p-code can be used to directly emulate instruction execution and generally follows the same control-flow, although it may add some of its own internal control-flow. The subset of opcodes that can occur in raw p-code is described in the section called "P-Code Operation Reference" and in the section called "Pseudo P-CODE Operations", making up the bulk of this document.

P-code is designed specifically to facilitate the construction of data-flow graphs for follow-on analysis of disassembled instructions. Varnodes and p-code operators can be thought of explicitly as nodes in these graphs. Generation of raw p-code is a necessary first step in graph construction, but additional steps are required, which introduces some new opcodes. Two of these, MULTIEQUAL and INDIRECT, are specific to the graph construction process, but other opcodes can be introduced during subsequent analysis and transformation of a graph and help hold recovered data-type relationships. All of the new opcodes are described in the section called "Additional P-CODE Operations", none of which can occur in the original raw p-code translation. Finally, a few of the p-code operators, CALL, CALLIND, and RETURN, may have their input and output varnodes changed during analysis so that they no longer match their raw p-code form.

4.2.2 Address Space

The address space for p-code is a generalization of RAM. It is defined simply as an indexed sequence of bytes that can be read and written by the p-code operations. For a specific byte, the unique index that labels it is the byte's address. An address space has a name to identify it, a size that indicates the number of distinct indices into the space, and an endianess associated with it that indicates how integers and other multi-byte values are encoded into the space. A typical processor will have a ram space, to model memory accessible via its main data bus, and a register space for modeling the processor's general purpose registers. Any data that a processor manipulates must be in some address space. The specification for a processor is free to define as many address spaces as it needs. There is always a special address space, called a constant address space, which is used to encode any constant values needed for p-code operations. Systems generating p-code also generally use a dedicated temporary space, which can be viewed as a bottomless source of temporary registers. These are used to hold intermediate values when modeling instruction behavior.

P-code specifications allow the addressable unit of an address space to be bigger than just a byte. Each address space has a wordsize attribute that can be set to indicate the number of bytes in a unit. A wordsize which is bigger than one makes little difference to the representation of p-code. All the offsets into an address space are still represented internally as a byte offset. The only exceptions are the LOAD and STORE p-code operations. These operations read a pointer offset that must be scaled properly to get the right byte offset when dereferencing the pointer. The wordsize attribute has no effect on any of the other p-code operations.

4.2.3 Varnode

A varnode is a generalization of either a register or a memory location. It is represented by the formal triple: an address space, an offset into the space, and a size. Intuitively, a varnode is a contiguous sequence of bytes in some address space that can be treated as a single value. All manipulation of data by p-code operations occurs on varnodes.

Varnodes by themselves are just a contiguous chunk of bytes, identified by their address and size, and they have no type. The p-code operations however can force one of three type interpretations on the varnodes: integer, boolean, and floating-point.

Operations that manipulate integers always interpret a varnode as a twos-complement encoding using the endianess associated with the address space containing the varnode. A varnode being used as a boolean value is assumed to be a single byte that can only take the value 0, for false, and 1, for true. Floating-point operations use the encoding expected by the processor being modeled, which varies depending on the size of the varnode. For most processors, these encodings are described by the IEEE 754 standard, but other encodings are possible in principle.

If a varnode is specified as an offset into the constant address space, that offset is interpreted as a constant, or immediate value, in any p-code operation that uses that varnode. The size of the varnode, in this case, can be treated as the size or precision available for the encoding of the constant. As with other varnodes, constants only have a type forced on them by the p-code operations that use them.

4.2.4 Pcode Operations

A p-code operation is the analog of a machine instruction. All p-code operations have the same basic format internally. They all take one or more varnodes as input and optionally produce a single output varnode. The action of the operation is determined by its opcode. For almost all p-code operations, only the output varnode can have its value modified; there are no indirect effects of the operation. The only possible exceptions are pseudo operations, see the section called “Pseudo P-CODE Operations”, which are sometimes necessary when there is incomplete knowledge of an instruction’s behavior.

All p-code operations are associated with the address of the original processor instruction they were translated from. For a single instruction, a 1-up counter, starting at zero, is used to enumerate the multiple p-code operations involved in its translation. The address and counter as a pair are referred to as the p-code op’s unique sequence number. Control-flow of p-code operations generally follows sequence number order. When execution of all p-code for one instruction is completed, if the instruction has fall-through semantics, p-code control-flow picks up with the first p-code operation in sequence corresponding to the instruction at the fall-through address. Similarly, if a p-code operation results in a control-flow branch, the first p-code operation in sequence executes at the destination address.

4.2.5 Control Flow Graph

cos’è e cosa fa

4.2.6 What we did

titolo da cambiare

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 the 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 object DecompiledResult with all the information needed. It is also possible to pass 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.

The graph 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 were intricate, and when it comes 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 needed to increase it to 500MB correctly decompile all the functions.

From the CFG files, we extracted 3 kinds of features:

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

4.2.7 Control Floe Graph unigrams complete

This first set of features contains the unigrams of the complete Pcode representation. The key for each feature is the concatenation of the Pcode, the input and output nodes. In particular, we construct the key as follow: `PCODE_nodeoutput#nodeinput*count` of nodes So this .json file is converted to:

`"pcodes": [ { "code": "CALL", "varnode_in": [ "ram", "const" ], "count": 2 }] key = call_ram#const*2` **Sistemare qua**

We counted the occurrences of each key and build our dataset.

4.2.8 Control Flow Graph Pcode only unigrams and bigrams

These two sets of features contain the unigrams and bigrams of the pcode only. We built the key using only the pcode operator, and then counted the occurrences. For the bigrams, we concatenated as before the key with the string =>.

4.2.9 Cyclomatic Complexity

Cyclomatic complexity is a software metric used to indicate the complexity of a program. Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of commands of a program, and a directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity may also be applied to individual functions, modules, methods, or classes within a program. The cyclomatic complexity of a section of source code is the number of linearly independent paths within it. For instance, if the source code contained no control flow statements (conditionals or decision points), the complexity would be 1, since there would be only a single path through the code. If the code had one single-condition IF statement, there would be two paths through the code: one where the IF statement evaluates to TRUE and another one where it evaluates to FALSE so that the complexity would be 2. Two nested single-condition IFs, or one IF with two conditions, would produce a complexity of 3. Mathematically, the cyclomatic complexity of a structured program[a] is defined regarding the control flow graph of the program, a directed graph containing the basic blocks of the program, with an edge between two basic blocks if control may pass from the first to the second. The complexity M is then defined as[2] M = E - N + 2P, where E = the number of edges of the graph. N = the number of nodes of the graph. P = the number of connected components.

The same function as above, represented using the alternative formulation, where each exit point is connected back to the entry point. This graph has 10 edges, 8 nodes, and 1 connected component, which also results in a cyclomatic complexity of 3 using the alternative formulation (10 - 8 + 1 = 3). An alternative formulation is to use a graph in which each exit point is connected back to the entry point. In this case, the graph is strongly connected, and the cyclomatic complexity of the program is equal to the cyclomatic number of its graph (also known as the first Betti number), which is defined as[2] M = E - N + P. This may be seen as calculating the number of linearly independent cycles that exist in the graph, i.e., those cycles that do not contain other cycles within themselves. Note that because each exit point loops back to the entry point, there is at least one such cycle for each exit point. For a single program (or subroutine or method), P is always equal to 1. So a simpler formula for a single subroutine is M = E - N + 2 Cyclomatic complexity may, however, be applied to several such programs or subprograms at the same time (e.g., to all of the methods in a class), and in these cases, P will be equal to the number of programs in question, as each subprogram will appear as a disconnected subset of the graph. McCabe showed that the cyclomatic complexity of any structured program with only one entry point and one exit point is equal to the number of decision points (i.e., "if" statements or conditional loops) contained in that program plus one. However, this is true only for decision points counted at the lowest, machine-level instructions.[4] Decisions involving compound predicates like those found in high-level languages like IF cond1 AND cond2 THEN ... should be counted in terms of predicate variables involved, i.e., in this example, one should count two decision points, because at machine level it is equivalent to IF cond1 THEN IF cond2 THEN [2][5] Cyclomatic complexity may be extended to a program with multiple exit points; in this case, it is equal to pi - s + 2, where pi is the number of decision points in the program, and s is the number of exit points.[8] **Citato da wikipedia, da modificare tutto**

Ghidra offers a class to compute the complexity of a function, CyclomaticCompelxity. 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. Then we calculate as a feature, the mean, the standard deviation, and the maximum value of complexity for each sample.

4.2.10 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.[9]

We exploit this functionality to determine which of the functions belongs to a standard library. Then we calculate the number of standard functions in the given sample and use it as a feature.

4.3 Rich Header features

Chapter 5

Classification and evaluation

5.1 Classification model

5.1.1 Validation

5.2 Features Selection

Chapter 6

Discussion

Chapter 7

Future works

...

Bibliography

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

[2] FireEye, "FireEye M-trends 2019." https://content.fireeye.com/m-trends.

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

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

[5] 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.

[6] M. Dubyk, "Sans institute," 2019.

[7] National Security Agency, "P-Code Reference Manual." https://ghidra.re/courses/languages/html/pcoderef.html.

[8] Wikipedia, "Cyclomatic Complexity." https://en.wikipedia.org/wiki/Cyclomatic_complexity.

[9] 0x6d696368, "Ghidra FID Generation." https://blog.threatrack.de/2019/09/20/ghidra-fid-generator/.