POLITECNICO DI TORINO

Faculty of Engineering

Master of Science in Computer Engineering

Master Thesis

Automatic Malware Signature Generation

Advisors
prof. Giovanni Squillero
Ing. Andrea Atzeni
Ing. Andrea Marcelli

Candidate:
Luca Cetro

October 2018

Acknowledgements

I would like to thank Professor Giovanni Squillero, Ing. Andrea Atzeni and Ing. Andrea Marcelli for the possibility they gave me of working on this project and for their assistance during the entire path.

Special thanks to Andrea Marcelli, who continuously and relentlessly followed the development of this work and with which I had the opportunity to exchange many ideas that ultimately were really helpful for this project outcome and for my professional growth.

I am thankful to all my friends and colleagues that have been a huge part of this entire university path and that were a big source of relief on hard times.

Finally, and most importantly, my biggest gratitude goes to my family, for the sacrifices they made for me to reach this goal, for their constant support and assistance all over this journey and with which I could share any positive and negative event, knowing they would have been there for me, no matter what.

It’s not because things are difficult that we dare not venture. It’s because we dare not venture that they are difficult.

Lucius Annaeus Seneca

Summary

Since 2010, the number of new malware released daily became so high that manual analysis is not an option anymore. In 2017 it was estimated that 120 million new malwares were released, that is about 200 per minute, mainly targeting the Microsoft Windows Operating Systems. Given the scale of the threat, emerged the need of automated approaches to detected new malware variants: several methodologies were proposed during the years, including a variety of machine learning approaches, although the signatures have been proved to be the most effective ones. A signature is a pattern that identifies a malicious code, and, even though they suffer from the so-called "specificity" problem, they have been demonstrated effective, scalable, and almost unaffected by false positives.
This thesis presents a research that aims to create a set of effective signatures, starting from a set of known malware samples, that are capable of matching new malware variants with very high recall, while reducing the number of false positives. The proposed procedure was implemented in a new framework, named YaYaGenPE, which targets the generation of signatures for Microsoft Windows executables, and produces YARA rules, the industry standard type of signatures. The designed framework was conceived as a pipeline consisting of three steps: the features extraction procedure, an optional clustering procedure and, finally, the rules generation step.
Several tests were performed using, as a comparison meter, available YARA rule generation tools. In particular three of them were selected: *yarGen*, *YaraGenerator*, and *yaBin*. Results show that YaYaGenPE generates less rules with respect to the other tools, with less false positives, comparable precisions and a better coverage of the training sets provided. Finally, some automatically generated rule sets were tested on the VirusTotal Intelligence RetroHunt service that applies the rules against a corpus of 100 TeraBytes containing benign and malicious software. Results show that, considering the huge amount of tested software, rules are accurate enough to avoid most of the false positives and targeted to specifically match samples of the family they were trained on or samples really close to the training ones.
The entire work has been done in collaboration with *VirusTotal*, one of the biggest malware analysis platforms, which, since January 2018, is part of Chronicle, a subsidiary of Alphabet Inc., the Google's parent company.

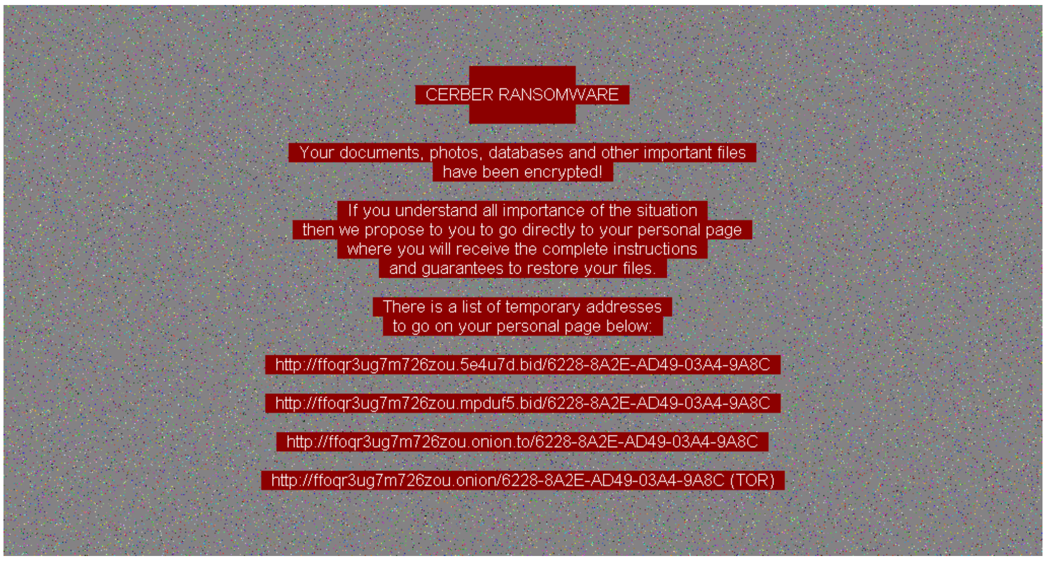
Contents

List of Figures

Chapter 1

Introduction

Malwares have been around since the early years of the IT area. Initially, these executables were not necessarily malicious, as they were prevalently diffused either for fun or for scientific purposes. Unfortunately, the era of malwares developed for fun is over, and, since 2000, malwares aim at wrecking havoc on the affected systems or, as a tendency begun by the 'Cryptolocker' malware, at locking all the machine's files asking for a ransom [8]. Figure 1.1 shows the screenshot of a victim machine infected by the "Cerber" ransomware, one of the most infective ransomwares for Windows 10 [44].



CERBER RANSOMWARE

Your documents, photos, databases and other important files
have been encrypted!

If you understand all importance of the situation
then we propose to you to go directly to your personal page
where you will receive the complete instructions
and guarantees to restore your files.

There is a list of temporary addresses
to go on your personal page below:

http://ffoqr3ug7m726zou.5e4u7d.bid/6228-8A2E-AD49-03A4-9A8C

http://ffoqr3ug7m726zou.mpduf5.bid/6228-8A2E-AD49-03A4-9A8C

http://ffoqr3ug7m726zou.onion.to/6228-8A2E-AD49-03A4-9A8C

http://ffoqr3ug7m726zou.onion/6228-8A2E-AD49-03A4-9A8C (TOR)

Figure 1.1: Screenshot of a victim machine infected by the "Cerber" ransomware. The victim is redirected to a set of (.onion) links which will give the possibility to pay the ransom.

Whatever the methodology used, it is evidenced, since 2010, the will of using malwares as an illegal way of making money or of denying victim's functionalities.

Although these types of software threaten both companies and privates, there is no doubt that the former are the ones suffering the most from this type of problems. A statement given by the cyber-security company "Carbon Black" in 2018 reported that approximately 92% of the companies they interviewed were interested by at least one cyber-security attack on the previous year [19].
The 2018 trend in Malware development, although slightly decreasing in the last 3 years, is still a major concern for IT security. The number of newly generated malwares is so high that, in 2017, it is estimated that 3.9 new malicious samples were generated per second [3]. Consequently, this should also be the ideal pace, for Anti-Virus software, of noticing and recognizing new malicious samples.
From an Operating System market point of view, out of the 121 million new malwares detected in 2017, statistical analysis show that Microsoft Windows and Android Mobile are the two preferred malware targets and the picture seems to be valid for year 2018 as well [3].

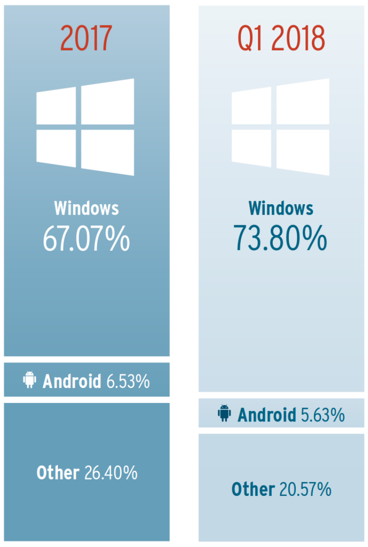


Figure 1.2: Operating System target percentages in 2017 and Q1 of 2018 (picture from [3]).

The previous facts evidence the impossibility, for the analysts, to manually analyze all the malicious samples that are constantly generated. On the other hand, it would ideally be possible, once determined the effective malicious nature of a sample, to create a unique signature that specifically identifies it, but the evolution rate of the malwares samples and the high variety of obfuscation techniques would make this approach ineffective, not to mention the huge database size that would be required to handle all the possible versions of the same executable.
For these reasons, it is needed an automated and effective approach towards malware detection, that is capable, possibly, of synthesizing several malicious samples in a restricted

amount of signatures, but at the same time, being not too generic to detect a high number of benign softwares as malicious (i.e., false positives), which might harm system usability. In particular, it would be ideal to have a set of signatures that maximizes the malwares coverage while keeping at minimum the number of benign executables erroneously targeted as malicious.
This thesis aims at satisfying this need of automated malware detection procedures. In particular, the main target of this work is the design of YaYaGenPE, a framework capable, given a set of malicious samples, of creating a set of signatures that cover all the samples, while at the same time identifying their relevant common traits.
The framework is focused on the Microsoft Windows Operating System executables, and it is a complement of the already developed YaYaGen framework [35][1], that targets the Android platform, with which it shares the signatures generation algorithms.
The signatures defined by both the tools are ultimately expressed in the form of YARA rules. YARA is a pattern matching tool whose success has been constantly growing since its creation, such that many cyber-security companies have been reported to extensively use it (e.g. “Kaspersky Labs”, “ESET”, “Symantec”, “Trend Micro” and many more) [67].
YaYaGenPE was conceived as a pipeline consisting of three steps: the features extraction procedure, an optional clustering procedure and, finally, the rules generation step.
The first step, i.e., the features extraction, relies on a specific set of static analysis features consisting of the Windows Portable Executable header fields, i.e., all the fields that identify the relevant parts of a Microsoft Windows binary executable. Static analysis features are, indeed, generally faster than dynamic analysis ones. Moreover, most of existing packers and obfuscation methodologies tamper the Import Address Table (IAT) and the executable entry-point only, leaving intact the majority of the PE header fields, while dynamic analysis features (i.e., executable’s behavior peculiar characteristics) are more complex and computationally intensive, sometimes also requiring human intervention.
The framework also supports an innovative set of features that are the user provided YARA rules. In particular, the procedure allows the user to provide some rules, that will be incorporated as features of the malicious samples when positively matching them. These rules might indeed be really helpful in detecting malicious samples and avoiding false positives, since they might be written by domain experts to specifically target designated malwares.
The second phase, i.e., the clustering, is an optional procedure and consists of two different algorithms. Working on the Windows Executables, it is indeed expected a big variety on the samples that the tool will have to handle. For this reason, YaYaGenPE supports, internally, two different clustering procedures that allow to reduce the size of the groups of samples the rules are generated on while working concurrently on the closest samples possible. This procedure is performed under the assumption that, working on groups of samples similar each other, the consequent signatures are capable of underlying the common features shared among them, without being too much generic (i.e., avoiding as much as possible the detection of benign samples as malicious).

The last step of the framework is the generation of a set of YARA rules for each cluster. Each rule is determined by finding an optimal combination of features that covers the highest number of samples possible, in the cluster. Two implementations are proposed: the “greedy” approach and the “clot” one. The former finds a set of rules by iteratively searching for the local optimal solution.
The latter is an improved version of the greedy approach, which produces overlapped rules, but approximately equally distributed in the number of covered samples, avoiding the problem of unbalanced rules in terms of numbers of literals.
Both the signature generation algorithms solve, in a sub-optimal way, the “Set Coverage” problem and, as such, are independent on the features provided, making the framework easily extensible to any new set of features in the future.

Chapter 2

Background

2.1 Malware

In computer systems security, malware is a common word to define any *malicious software*: a software that harms a user, a target system or an entire network.
The first hypothesis of malware existence was introduced by John von Neumann, who, by 1951, theorized the possibility of self-replicating automatons [33]: this definition is what, nowadays, identifies *viruses* and *worms*.
During the years, new outcomes were produced on the same topic: in 1959, Lionel Penrose presented an automated self-replication model, which was then developed by Frederick G. Stahl on an IBM 650.
The first recognized virus was publicly demonstrated by Fred Cohen in 1983, while he was a graduate student at USC. Then, in 1986, two Pakistani programmers created the Brain virus, which was capable of escaping detection by simulating the system calls that were, at the time, used as a signature of virus [17].
From that point on, the spreading of malware has been constantly growing, also due to the diffusion, in the wild, of automatic tools to generate variants of the same program (these tools are usually known as Mutation Engines).
Early days malwares were initially created either for fun or for scientific purposes [33]. However, as the knowledge on computer systems grew up among the community and with the uprising success of Internet, several authors started creating malwares for profit.

2.1.1 Malware typologies

During the years, several different typologies of malwares have been created. All of them try to harm the system in a way or another, however, different typologies differ for their different attack methodologies.
The most famous malware classes are [37]:

- *Backdoor*: malicious code that allows unauthorized and unauthenticated access to the attacker.

- *Botnet*: similar to a Backdoor, but involves several infected hosts, all executing the same set of instructions given by the attacker(s) (this methodology is also known as Command-and-control).
- *Downloader*: a malicious code that is used to download malicious software from the Internet and execute them. This type of malware is installed by attackers the first time they get access to the system.
- *Information-stealing malware*: this category of malware is used to gather informations from the infected systems and convey them to the author. There are several types of information-stealing malwares, such as sniffers, password hash grabbers and keyloggers, however, the common goal of any of them is typically to retrieve access credentials of the victim for any given service she signs into.
- *Launcher*: malicious program used to launch other malicious softwares. This is typically used to ensure stealth or higher level access permissions to a system.
- *Rootkit*: this type of malware comprehends several other malicious codes, one of which is typically a backdoor, and it is used to conceal them and to give control of all the possible softwares to the attacker.
- *Scareware*: software designed to scare the victim, typically by telling that the system is infected, in order to convince her that buying the recommended software would solve the problems. In reality, what the bought software does is to clean-up the system from the scareware.
- *Ransomware*: also considered as part of the Scareware category, it slightly differs from the previous class' behaviour: Ransomwares encrypt relevant parts (typically user data) of the operating system or even the entire hard drive and blackmail the victims to pay the attacker in order to get the decryption key [27].
- *Spam-sending malware*: a malware that infects a system and uses it as a vector to send spam e-mails to other systems.
- *Viruses*: a particular type of malware that damages the infected system and then uses the network connection to replicate on near and/or reachable systems;
- *Worms*: they use an approach similar to the one employed by viruses, however, they damage the machines in which they are injected by saturating the resources of the hosting machines through constant self-reproduction;
- *Cryptocurrency-mining malware*: one of the latest malware trends. It is a particular type of malware that uses the victim's machine in order to mine crypto-currencies for the attacker [26].

2.1.2 Malware evloution

During the arms race between malware authors and analysts, the structure and the obfuscation methodologies used in malicious executables changed rapidly. Initially, the machine code was completely unprotected and this allowed analysts to easily target opcode sequences to recognize specific malware families. To evade such signatures, malware authors employed several obfuscation techniques.
Obfuscation techniques were originally introduced to protect intellectual property of software, by generating new executables that behave like the old ones, but structurally different [69], eventually even harder to disassemble and to analyse.

Encrypted malware. The first category ever introduced of obfuscation was the encryption of the malware body. This methodology allowed authors to hide the content of the malware and, as a consequence, to avoid signature matching. Indeed, after a malware has been encrypted, the whole binary code is hidden and it does not match an eventual signature previously generated over a plain sample. Moreover, by simply changing the encryption key from generation to generation, also the encrypted content changes, thus, it was not possible to create a simple signature over the entire malware. However, in order to recover the original executable code, this type of obfuscation requires a decryption stub, which, in the early days, was remaining constant across different generations. Because of this, malware analysts started creating a signature of the decryption stub instead of the entire malware body.
Early days encryption/decryption stubs were also really simple and efficient, ultimately, as simple as an XOR cipher [49].

Oligomorphic Malwares. The next step, in order to avoid the just mentioned signature, was to create a methodology capable of generating a different decryptor at each new generation. The first attempt in this direction consisted in the so called *Oligomorphic* malwares, that were capable of creating a few hundred different decryptors. Nevertheless, the number of generated decription stubs was still manageable through decryptor signatures matching.
The first example of oligomorphic malware was the Whale virus, detected in 1990. This executable contained a few dozen decryptors embedded and, when spreading to other machines, it chose randomly one among these decryptors and the correspondent encryptor to create the malware variant [49].

Polymorphic Malwares. As few hundreds of decryptors were not enough to thwart signature matching analysis, malware authors created tools that aimed at the generation of countless number of decryptors, through the usage of several obfuscation techniques (some of those will be described in the next sections). One tool of the kind was *"The Mutation Engine (MtE)"* [69]: this engine was capable of converting a plain malware to a

polymorphic one by linking the engine to the malicious executable [49].
This new obfuscation methodology was a huge problem back in the days when it came out, however, anti-virus vendors started analysing the suspicious samples by actually executing them in a protected environment (i.e., sandbox). During the execution, the analysts can then keep track of suspicious behaviour or, eventually, extract the machine code of the malware once the decryptor has done its job.
Even though this last methodology works, it is not always effective due to *armoring* techniques: a set of techniques that deny emulators to properly track the behaviour of the executable (e.g., *anti-debugging* and *anti-Virtual Machine* techniques [37]).

Metamorphic Malwares. The last advance in malware obfuscation was the metamorphism. This particular category of malware variants generation directly targeted the malicious code of the executable, so to make it equivalent from a behavioural point of view, but structurally different from time to time. As opposed to the polymorphism, in which the changings were performed to the decryptor, by adding some of the typical obfuscation techniques, metamorphism directly target the code responsible of the malicious behaviour, that, in this way, cannot be used anymore to generate a signature once decrypted. In particular, if properly designed, metamorphic malwares are capable of creating millions of functionally equivalent variants, from which it is not possible to find a common pattern.
The first example of metamorphic malware was the Win95/Regswap virus, found in 1998 [50]. This virus did not use polymorphic decryptors, instead, it generated variants of its body by keeping a constant operation flow but changing the registers involved in the operations. This technique, also known as register swapping, is one among several metamorphic transformations.
Other remarkable examples of metamorphic malwares are:

- *W95.Zmist*: released by the famous malware author Z0mbie, this malware used most of the metamorphic transformations and even a polymorphic decryptor. The value of this sample resides in the “Mistfall engine”: an engine capable of decompiling target executables into a set of objects, separately mutate each of them, insert itself in between the objects, and put all the pieces together in a new sample.

- *Simile*: found in 2002, this virus was famous thanks to the “MetaPHOR engine”, which accounted for almost 90% of its assembly code [56]. MetaPHOR is an acronym for Metamorphic Permutating High-Obfuscating Reassembler, and, while performing metamorphic mutations, it allowed virus to grow or shrink in size. Moreover, this engine was capable of properly infecting also Linux’s ELF executables [50], even if the original virus was designed for a Windows environment [56].

2.1.3 State of the Art on Malware Analysis

As malware started spreading all over the world, analysis techniques have been developed by the researchers in order to detect whether an executable is malicious or not.
During the years, three types of analysis have been defined [62]:

- *static analysis*;
- *dynamic analysis*;
- *hybrid analysis.*

Static Analysis techniques

The static analysis techniques try to detect malwares *without* executing the analysed sample [23]. These methodologies focus on the peculiar features of the malicious samples with respect to the benign ones.
This family of techniques is generally the fastest and the safest one, since it does not require running the samples in a protected environment. However, it has been proved that, when using obfuscation techniques, static analysis are usually not enough to identify malicious samples, and, to make them obfuscation resilient, they may lead to obfuscator detection (and, as a consequence, to false positives) [38]. Moreover, some static techniques, such as those hash or string based, are targeting the reactive detection of malicious samples, i.e., they are typically not capable of detecting zero-day malwares since they are based on signatures derived from already analysed samples.
Nevertheless, this approach is still one of the most widespread, due to the limitations (in terms of computational time and behavior detection) of the dynamic counterpart.
Some of these techniques try to detect string signatures, byte-sequence n-grams, syntactic library call, control flow graph analysis and opcode frequency distribution.
Some other approaches try to analyse the header structures of the executable samples: the basic idea of the just mentioned analysis is to try to find peculiar header fields that distinguish malicious executables from benign ones.

Case of study: PE-probe

As an example of the last static analysis category, PE-Probe uses a selection of the most relevant features of a PE Executable to detect when a sample is malicious [54]. This framework is an evolution of the PE-Miner one, which aimed at detecting relevant features of the PE header and feeding them to a set of classification algorithms in order to find the best malware classifier [55].
The basic idea of PE-Miner is to:

1. select a set of features from the PE header, such as imported DLLs, linker versions and imported resources numbers;

2. perform a preprocessing step, which involves features reduction (using redundant feature removal, principal component analysis or Haar Wavelet Transform);

3. feed the classifier with the identified samples (they used different types of classifiers and detected that the decision tree J48 was the best in terms of accuracy).

They claimed, using a combination of each classifier with each feature selection methodology, AUC [1] values not lower than 0.936, which is fairly high, considering the fact that, as they mentioned, approximately 40 % of the samples were packed and around 12.5 % of the remaining ones were detected as being not packed nor unpacked, meaning that they were likely obfuscated through a custom packer (which is not detected by PEiD or Protection ID [2]).
PE-Probe is an improvement of the PE-Miner framework and tries to overcome the limitations it had. In particular, the authors claimed that, when training PE-Miner using only packed executables and then testing with non-packed ones, the accuracy decreases of approximately 10 %.
Because of this, PE-Probe adds a preliminary non-signature based classification to detect which samples are packed and which not. Starting from that outcome, a set of relevant features is extracted per class, in order to adapt to the two different situations, and then the resulting samples are given to the second step of PE-Miner.

Dynamic analysis techniques

As opposed to the static analysis techniques, the dynamic ones try to detect malicious behaviour by actually running the executable in a controlled environment and tracking the operations that are performed. This procedure, however, requires a safe environment (sandboxing or virtual machines) and a set of monitoring tools to keep track of file system, registry and networking activities [23].
The dynamic analysis is more effective than the static one in terms of behaviour detection, since it does not need the executable to be disassembled. However, it is computationally

[1] AUC is an acronym for Area Under the Curve, which is used to evaluate the quality of a binary classifier. Specifically, considering the True Positives ratio (TP) on the x-axis and the False Positives ratio (FP) on the y-axis, it is possible to draw the curve of a binary classifier in terms of TP and FP by varying one of its parameters. The Area Under the Curve is then a measure of the portion of the graphical area under the drawn curve. The AUC values range from 0.0 to 1.0, where values around 0.5 indicate an almost randomic classifiers, while extreme values (i.e., values near to 0.0 and to 1.0) indicate a good classifier. Interestingly, classifiers with an AUC close to 0.0 are really good in guessing the opposite of the correct classification, which, opportunely reversed, still identify good classifiers.

[2] PEiD and Protection ID are two of the most famous signature based packers publicly available.

expensive, time consuming and it is not always effective. In fact, during the years, several anti-debug and anti-virtual machine techniques [37] have been implemented by malware authors in order to prevent malicious behaviour to be tracked (review of some of these methodologies in the following sections). From what it has just been said, it is clear that neither static analysis nor dynamic one are perfect, and that the two of them should be complementary.

Hybrid analysis techniques

This last set of techniques is just an ensemble of the previous two. The motivation of these is in the last line of the previous subsection: static and dynamic analysis are complementary each other.
Hybrid analysis aims at having a complete view of the malware's behavior by bypassing obfuscation and anti-analysis techniques through the combined usage of static and dynamic analysis. Indeed, most of the times, static disassembler are not capable of coping with anti-disassembly and/or packing techniques, while dynamic analysis might not be able to cover all the code paths of the executable, due to trigger conditions and/or anti-debugging or anti-virtualization techniques.
Some practical applications have been developed using hybrid analysis, both for malicious behaviour detection and for complete and resilient code disassembly. In [47], the authors created a framework that combines static and dynamic analysis to create a complete control-flow graph that is returned to the analyst in order to help her studying the samples. The application uses several countermeasures, introduced in several previous works, to anti-analysis techniques (both static and dynamic), allowing to merge the contribution of the two analysis in an overall graph. The authors also claimed support of several commercial packers (such as UPX, ASPack, PECompact and others) with good coverage results.

Case of study: HDM-Analyser

As an example of hybrid analysis targeting malware detection, it is here briefly described HDM-Analyser [20].
The tool aims at creating a classifier from features that were extracted from *non-packed* executables through hybrid analysis. In particular, the features that have been selected in that work were the sequence of called APIs [3], since they have good chances of tracking the malwares behaviour. As it will be described in the next section, static analysis might not be enough to extract the API calls, due to anti-analysis techniques, so, the approach

[3]API stands for *Application Programming Interface* and indicates the set of library functions that the OS and/or the programming language put at programmer's disposal to perform some tasks.

chosen was to rely also on dynamic analysis. However, as the authors pointed out, one of the main disadvantages of the dynamic analysis is that it is time consuming since it requires the execution, at testing time, of the analysed sample. On the other hand, a small overhead during the training phase might be acceptable in return of an higher accuracy in the feature extraction. The aim of the framework is that of using dynamic analysis as a heuristic to guide the static disassembler on the choice of the paths to follow in any decision point during the testing of a sample.
Specifically, when there's a branch instruction (e.g., JZ [EAX]), common disassemblers are not usually capable of detecting which will be the outcome of the branch, so they will limit to disassemble entirely the code. However, in order to extract the API call sequences it is required to understand which direction will be taken by the branch. At testing time, without dynamically executing the sample, this is not possible; because of this, the authors applied a machine learning technique (in particular, a Bayesian Netowrk) to provide which will be the most likely outcome of the branching instruction without executing it, but only looking at a window of n instructions preceding the branching one (experimentally, n was set to 3).
During the training phase, the static analysis part creates an "Enriched Control Flow Graph", that is a control-flow graph in which each node represents a jump/branch instruction (or eventually a call/ret instruction through a proper substitution) and each edge represents a target instruction or the API spanning (an API is identified by an edge that goes from the instruction pointed by a CALL to the first RET instruction). Every node is also associated with an enrichment vector, which contains a counter for each type of instruction, in a simplified instruction set, that indicates how many times that instruction occurred in the n instructions preceding the node's one.
Meanwhile, the dynamic analysis creates the actual control flow of the sample. Once both analysis have been performed, the control flow graph is traversed, node by node, and, at each edge, it is checked whether that edge represents the API call in the sequence of APIs identified by the dynamic analysis. If the edge represents an API call, there are two possibilities:

- if the node's taken edge refers to the next instruction in the flow, the node is labeled as negative (the jump instruction is not executed);
- if the node's taken edge refers to a jump instruction, it is labeled as positive.

All the other nodes, that were not associated with an API edge remain unlabelled. At the end of this procedure, all the labeled nodes of the graphs of the executables are collected in an "Enrichment Table". This table will then be used to construct the Bayesian Network. Using that Network to define API calls sequence at testing times showed results that were intermediate in between dynamic analysis and static one, both in terms of accuracy and in terms of execution time. In particular, the authors claim that the framework is capable of reaching accuracies higher than the static analysis with only a small execution time overhead.

2.1.4 Malware obfuscation techniques

This section provides a brief overview of some of the most famous obfuscation techniques.

Dead-Code Insertion. One of the simplest obfuscation techniques to change the appearance of an executable is to add sequences of instructions whose effect, on the behaviour of the code, is negligible. The easiest way to do this is to add a sequence of NOP instructions, i.e., instructions that are used just to waste CPU clock ticks.
However, the previous example is so easy that antivirus scanners can recognize and remove them from the executable before computing the signature. Because of this, malwares tend to use several other irrelevant instruction sequences that are not that trivial to detect.
As an example, lets assume a piece of code needs to push the register EAX onto the stack. The straightforward way to do this is by:

```
PUSH EAX
```

However, this simple instruction could be easily written also as:

```
NOP
NOP
DEC EAX
PUSH EAX
INC BYTE PTR SS:[ESP]
INC EAX
```

which is a much more complex way of having the same result. This simple substitution, however, will prevent the signature of an executable that uses the latter to match the one of an executable using the former.
A similar approach is known as *Do-little* code: this methodology, instead of inserting useless code, adds an heavily obfuscated set of operations whose result will be of some utility for the next instructions. In this way, analysts cannot remove this piece of code and, at the same time, the obfuscations implied will slow down the code interpretation [46].

Register Reassignment. This obfuscation technique was originally found in the Win95/Regswap virus. The technique consists in using the same set of logical instructions of an executable but performing them using, as operands, a different set of registers from generation to generation.
Although the virus code changes, the variation is not that high, so, viruses can be still detected by usign wildcard characters in signatures [48].

Instruction Reordering. The basic idea of this technique is to keep the flow of instructions constant, but to rearrange the body of the executable by swapping blocks of machine code. In particular, metamorphic viruses divide the code into a set of non-overlapping blocks, then, they shuffle the order of these blocks in the executable, but introduce unconditional jump instructions accross blocks so that the execution order is still the original one.

Instruction substitution. This obfuscation technique exploits the possibility of performing the same operation with different machine code instructions. As an example, lets assume it is needed to set register EAX to zero; in order to do this, there are several different possibilities, such as:

1. MOV EAX,0
2. XOR EAX,EAX
3. SHR EAX,32
4. SHL EAX,32
5. AND EAX,0

and many others. Metamorphic viruses use the same principle in order to create different variants of the same executable.

Instruction transposition. In this technique it is searched for sequences of independent instruction chunks. Basically, each chunk of instructions (at least one instruction per chunk) cannot be split and reordered, being composed by dependent instructions, but its position is irrelevant with respect to the position of other chunks. Because of this, it is possible to safely shuffle the order of the chunks in the executable without altering the behaviour of the program.
For example, if a program executes the following code:

```
MOV EAX, EDX
INC ECX
```

the two instructions (i.e., chunks) are independent from each other. Because of this, it is possible to generate an executable variant, equivalent to the original one, by reverting the instructions order:

```
INC ECX
MOV EAX, EDX
```

Code Integration. This technique was introduced by the Win95/Zmist malware. That malicious software decomposes a target executable into a set of objects and inserts itself in between them, by making sure that, after recombining all the pieces together, everything still works properly. This obfuscation methodology is one of the most sophisticated and it may make detection really hard [69].

2.1.5 Anti-disassembly techniques

These methodologies use ad-hoc crafted code and/or data to trick disassembly tools into producing a wrong disassembled code. As a consequence, the usage of these techniques by malware authors implies a time-consuming analysis from the malware analyst, eventually even preventing the source code to be retrieved in a reasonable amount of time. Moreover, these techniques might also obstacle automated and heuristic analysis tools that rely on disassembled code [37].
The basic idea behind anti-disassembly techniques is to exploit weaknesses in disassembler algorithms. Specifically, each disassembler works under some assumptions over the executable to be disassembled, so, when these assumptions are not verified, the tool is not capable of returning the correct code.

Disassemblers

There exist two types of disassembler: linear and flow-oriented disassembler.

Linear Disassembler It is the simplest disassembler among the two and also the easier to trick. This tool works under the assumption that assembly instructions are linearly organized and one instruction begins immediately after the termination of the previous one. As a consequence, the disassembler identifies a new instruction by taking the length of the previously parsed instruction and adding it to the beginning of that instruction's bytes.
This technique has some problems when dealing with code that contains instructions and data in the same bytes sequence. This might be the case, not only of malicious executable, but also of PE executable files: typically, PE-executables contain a single code section called ".text", but, at the end of this, there might be a jump table containing a list of pointers. Thus, this table will not contain instructions, but only offsets to other locations. However, linear disassemblers are not capable of detecting where does the code stops and the data begins, so, they interpret that jump table as a set of instructions.

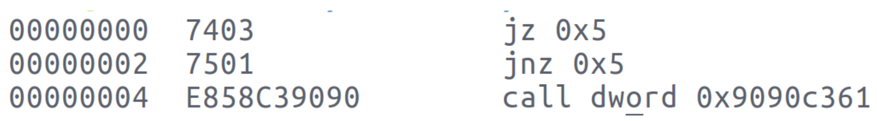
Flow-Oriented Disassembler As opposed to the previous category, flow-oriented disassemblers examine each instruction and keep track of the flow of the code. This means that, if there was a JMP instruction at some point in the code, this disassembler does

not blindly parse the bytes immediately following the JMP instruction's ones, but directly disassemble the bytes at the jump destination address.
Although this behaviour is more resilient than the linear disassembler's one, there are still some issues, in particular when dealing with conditional branches.
In those cases, the disassembler has to follow both the true branch and the false one (typically starting from the false one), however, when dealing with handwritten code, and, specifically anti-disassembly code, this behaviour might lead to some problems.

Anti-disassembly techniques

The following lines will describe briefly the mechanisms of the simplest techniques to trick linear and flow-oriented disassembler (at least when not assisted by human analysts).

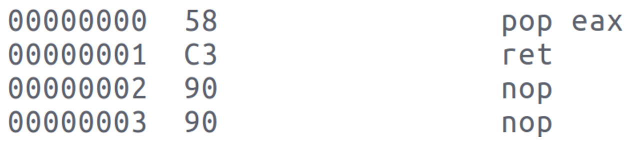
Jump instructions with the same target. This technique consists in consecutively putting two complementary jump instructions pointing to the same target address, which, overall, mimics the behaviour of an unconditional jump instruction.



```
00000000  7403                jz 0x5
00000002  7501                jnz 0x5
00000004  E858C39090          call dword 0x9090c361
```

Figure 2.1: Example of jump instructions to same target.

Figure 2.1 represents an example of this technique. In this example, both instructions jump to the address 0x5, however, address 0x4 contains the byte 0xE8, which is the beginning of a "CALL" instruction. Because of this, a linear disassembler interprets the following bytes as a single instruction; however, when starting to disassemble at address 0x5, the resulting instructions are those in Figure 2.2. The additional byte that causes these misalignments is commonly known as *rogue byte*.



```
00000000  58                  pop eax
00000001  C3                  ret
00000002  90                  nop
00000003  90                  nop
```

Figure 2.2: Disassembled code of Figure 2.1 starting from the target address.

This technique is also capable of tricking flow-oriented disassemblers, since these disassemblers consider the two jump instructions separately and, usually, start parsing the code from the false branch.

Jump Instruction with Constant Condition. Another frequent technique is the usage of conditional branch instructions, which, ultimately are equivalent to unconditional jumps. This could be easily implemented by fixing the tested bit just before the testing instruction, so that the condition will always take one direction and never the other one (Figure 2.3).

```
00000000  33C0                 xor eax,eax
00000002  7401                 jz 0x5
00000004  E958C39090           jmp dword 0x9090c361
```

Figure 2.3: Example of jump instruction with constant condition.

As shown in Figure 2.3, disassemblers are not capable of detecting the jump to offset 0x5, so, they will disassemble linearly, starting from the rogue byte immediately following. The same thing happens also for flow-oriented disassemblers, which, usually, start disassembly from the false branch and trust it as valid. Actually, when performing the disassembly starting at address 0x5 the result is the one in Figure 2.2.

Impossible Disassembly. This technique is much more complex than the previous two, and has several different application possibilities. Its basic principle is somehow the opposite of the rogue byte one: there is a sequence of bytes that is part of multiple instruction flows. As an example, lets refer to the opcode string in Listing 2.1

```
EB FF C0 48
```

Listing 2.1: Impossible Disassembly opcode string.

When this code is disassembled, the output is something like the one in Figure 2.4.

```
00000000  EBFF                 jmp short 0x1
00000002  C0                   db 0xc0
00000003  48                   dec eax
```

Figure 2.4: Example of erroneous disassembly of Listing 2.1.

By taking a look at Figure 2.4, the first two bytes of Listing 2.1 represent a jump instruction to the absolute address 0x1, which is the byte “FF”. This means that the “FF” byte will be part of two instructions: “EBFF” and “FFC0”. In particular, the code executed after the jump is in Figure 2.5.

As it is easy to see putting the pieces together, what this code is doing is just a complex NOP instruction. However, a similar approach could be used for much more complex situations.

The function pointer problem. This technique uses function pointers to reduce the information about program flow that can be extracted. In particular, if function pointers are used in handwritten assembly or crafted properly, it might not be possible to reverse-engineer the executable without the usage of dynamic analysis.

```
00000000  FFC0              inc eax
00000002  48                dec eax
```

Figure 2.5: Code executed after the jump of Listing 2.1.

Return Pointer Abuse. The idea of this technique is to confuse the disassembler by using the RET instruction when it shouldn't be used. Using a RET instruction can have two results: either the disassembler is not capable of showing code cross-reference of the target instructions or, in the worst case, the disassembler stops its execution, believing the end of the program was reached.

Misusing Structured Exception Handlers. The Structured Exception Handling is a mechanism that allow to manage error conditions without abruptly terminate the program. This feature, although useful, it is not always supported by disassemblers, so, it is easily usable to trick disassemblers. Since it is not so complex to create a personalized exception handler on top of the chain already in the stack, a malware author could create a new exception handler and, through an operation that is guaranteed to throw an exception (such as a division by zero), jump directly on an external routine. This routine could eventually be parsed properly by the disassembler, however, there is no indication that it is referenced by any piece of code.

2.1.6 Anti-debugging techniques.

This set of anti-analysis techniques is used to thwart analysis that relies on debugging to understand the malware's behaviour. There is a big variety in the techniques that can be used for this purpose, in the following paragraphs there will be introduced some of them.

Windows Debugger Detection.

This particular category of anti-debugging techniques is strictly based on the Microsoft Windows environment and on the utilities it offers.

Using Windows API. The easiest of these techniques is the one that relies directly on the Windows API to understand if a program is being debugged. Windows directly provides a set of functions to detect debuggers, however, there are some other functions that were designed for different purposes, but that have been adapted to debugger detection. Some of the previously mentioned functions are:

- IsDebuggerPresent: as the name says, this function detects the presence of the debugger. In order to do this, it relies on the Process Environment Block[4] structure and

[4]A Windows Process Environment Block (PEB) is a data structure, kept by the OS, that, for each

checks the “BeingDebugged” field, which, if debugged, should be a nonzero value.

- CheckRemoteDebuggerPresent: this function behaves exactly as the previous one, with the exception that it has the possibility of monitoring, not only if the current process is being debugged, but also if any other process in the local machine is, provided that its process handle is known at runtime.
- NtQueryInformationProcess: this function can be used in order to retrieve informations about a given process. It receives a process handle as first parameter and the type of process information requested as second one. In order to understand if the process is being debugged, it is then sufficient to call the function on the current process' handle and, as second parameter, the value “ProcessDebugPort”. If the process is not debugged, NtQueryInformationProcess will return zero, otherwise a nonzero port number.
- OutputDebugString: this function sends a string to the debugger for display. However, this function can be used to test if there actually is a debugger receiving that string. In particular, a process could try to send a string to an eventual debugger by OutputDebugString and then check if the GetLastError function indicates that an error occurred. If the function indicates no errors, then the process is being debugged, since it means that the OutputDebugString call was successfull.

Manually Checking Structures. Using the Windows API is a practical way of understanding if the process is being debugged, however, malware authors tend to prefer manual checking methodologies. The reason behind this is that these APIs could be hooked by a rootkit to return false informations, so to trick the anti-analysis checks. Among the several possibilities for manual checks there are:

- Checking the BeingDebuggedFlag. This technique emulates manually what was performed by the Windows API IsDebuggerPresent function.
- Checking the ProcessHeapFlag. In this approach, it is searched for two particular fields (the ForceFlags and the Flags fields) in the ProcessHeap. A ProcessHeap is a structure located at offset 0x18 in the Process Environment Block and contains an header in which, on the just mentioned fields, it is specified whether the heap was created within a debugger or not.
- Checking for NTGlobalFlag. Processes tend to create memory heaps differently when they are started within a debugger. In particular, at offset 0x68 of the PEB it is

running process, stores all the user-mode parameters associated. In this context, what is relevant is that, among other fields, this structure contains a BYTE “BeingDebugged” that indicates if the process is being debugged or not.

specified information about how to create the heap. If, at that offset, there is the value 0x70, it means that the process is being debugged. Actually, the value 0x70 is a combination of 3 different flags, that are set if the process is started by a debugger.

- Checking for system residue. This technique searches on the OS for traces left by a debugger. Examples of traces can be: registry keys modified by debuggers, files or directories associated with debugger executables or in-memory debugger traces.

Identifying Debugger Behaviour

Debuggers are usually used by setting breakpoints or stepping forward instruction by instruction. These debug modes, however, modify the original process' code. This particular class of anti-debugging methods aims at detecting these kind of operations. Examples of these techniques are:

- Scan for INT 3 instruction. The INT 3 instruction is a particular software interrupt used by debuggers to trigger the debug exception handler and to stop execution (that is a basic way of setting a breakpoint). The opcode of this instruction is 0xCC. Malwares could, at the beginning of the execution, through a CALL and POP sequence of instructions, retrieve the portion of source code being executed and search for that particular opcode. If the opcode is found, then it means that, if things have been designed properly, the process is being debugged.
- Performing Code Checksums. This technique resembles somewhat the previous one, however, instead of searching for the 0xCC opcode, it is performed a cyclic redundancy check (CRC) or a checksum of the executable code and it is compared to the original checksum. If the two checksums do not match it means that something (in this case a debugger) has modified the executable code.
- Timing Checks. This technique exploits the time delay required when an executable is run under the supervision of a debugger, in particular when the debugger is going through a step-by-step execution. However, in all cases, there are two main methodologies to realize if a process is being debugged:
  - get a timestamp, execute some instruction and then get another timestamp. If the process is being debugged, the execution will take more than when not, so, it is possible to detect a debugger.
  - get a timestamp before and after raising an exception. This technique is relying on the fact that, when a process is debugged, exception handling usually requires human intervention to prosecute, which introduces a huge delay. Some debuggers allow to ignore exceptions, however, even in these cases, the ignoring process requires some time delay.

2.1.7 Anti-Virtual Machine techniques

The last type of analysis of malicious samples is the one relying on Virtual Machines. With this approach, analysts build up a protected Virtual Machine environment in which the malware is free to execute and perform its malicious tasks. Malware authors, as a consequence, have developed a set of techniques to detect and evade Virtual Machine analysis. Most of these techniques target a specific Virtual Machine, that is VMWare.

VMWare Artifacts. Generally speaking, each Virtual Machine is known to leave artifacts on the installed Operating System. Malwares can use these artifacts to know if they are running in a virtual environment. The following techniques take, as an example, the VMWare Virtual Machine artifacts. There is a set of artifacts that can be used for this purpose, such as:

- search for running processes related to the VMWare Tools Service;
- search for registry keys related to VMWare, typically related to virtualization of the hardware;
- check the MAC address: each MAC address has the first 3 bytes related to the vendor of the network interface. However, if the process is running on a Virtual Machine, also the network interface will be virtual, and it will be associated to VMWare as a vendor. In this case, the first 3 bytes will be 00:0C:29.
- search for hardware versions: when virtualized, the hardware versions, such as those of the motherboard, will be associated to the Virtual Machine.

Checking for memory artifacts. This approach is similar to the one used in the anti-debugging techniques, i.e., it is searched, in memory, for artifacts left by the Virtual Machine.

Vulnerable Instructions. This last set of techniques relies on particular instructions, which, when executed on a virtual machine, have an output that is different with respect to the one that would have been returned if executed directly on the hosting machine.

2.1.8 Packers

Packers were initially developed, in the 80's and early 90's, to compress the executables in order to reduce the size they occupied both in the secondary memory and in RAM [50]. These tools consist of a packing stub, typically not included in the generated executable, and of an unpacking stub. The former is in charge of compressing and (eventually) encrypting the executable body, while the latter reverses the operations and restores the original body at runtime.

Packers, together with compression and encryption, typically include several of the previous obfuscation and anti-analysis techniques, ideally to provide intellectual property protection. As always, these capabilities have been employed not only by legitimate authors, but also, and prevalently, for malicious purposes.

Detecting packed samples

Packed samples can be detected with a certain degree of confidence through the Shannon Entropy measure, however, this is just an heuristic approach. In addition, this measure allows to know, with a certain probability, *if* an executable is packed or not, but it does not tell anything about *which* is the packer used. To address this last problem, there are packer signature databases. Nevertheless, tools that use such databases (as PEiD or Sigbuster) are not always capable of matching packers due to the high variety in the wild and due to the evolution and changes in packers' codes. Moreover, it might happen that malware authors modify some of the code related to the packers so that the generated samples do not match any packer signature.

Packers complexity and diffusion

Packers, in the market or custom made, have different characteristics. The different features and unpacking methodologies used define several levels of complexity. Understanding packers complexity is not a trivial task, however, some studies have been made to try to give it a standardized classification. The study in [61] identifies 6 types of packers, having increasing complexity.
Packers ranging from type 1 to type 5 allow, at some point in time during execution, to have a complete view over the unpacked malicious code, while they only differ for the number of decryption routines applied and for the obfuscation methodologies included. Type 6 packers are the most complex ones, for which only a slice of code is unpacked in memory from time to time. This means that, in order to get the complete code, it is necessary to take several memory dumps at different time instants.
Most notable examples of commercial packers are:

- **UPX**: it is a Type 1 packer, so, the simplest category. It consists of a single decryption routine, at the end of which there is a transition to the original executable;
- **UPolyX**: it is an UPX scrambler, that operates on the output given by UPX. It consists of a polymorphic engine with two interleaved unpacking layers, the outermost of which extracts part of the original code plus the innermost unpacking layer. As a result, this packer is a Type 3;
- **Armadillo**: this packer is one of the most complex ones. It is a Type 6 packer consisting of two processes, the father, in charge of unpacking the code, and the

child that executes the original program.

- **VMProtect** and **Themida**: those two packers, among other possibilities, allow for embedded virtual machine packing. In this particular methodology, they encode the original executable into a purposely created machine language, and add, in the packed executable, a virtual machine capable of correctly interpreting it.

2.2 Automatic Approaches to Malware Analysis

The number of newly discovered malwares per year has shown, in the last 10 years, an exponential growth [2]. Figure 2.6, although referring to 2012, perfectly represents the situation perceived by Anti-Virus companies about the amount of threats yearly created.

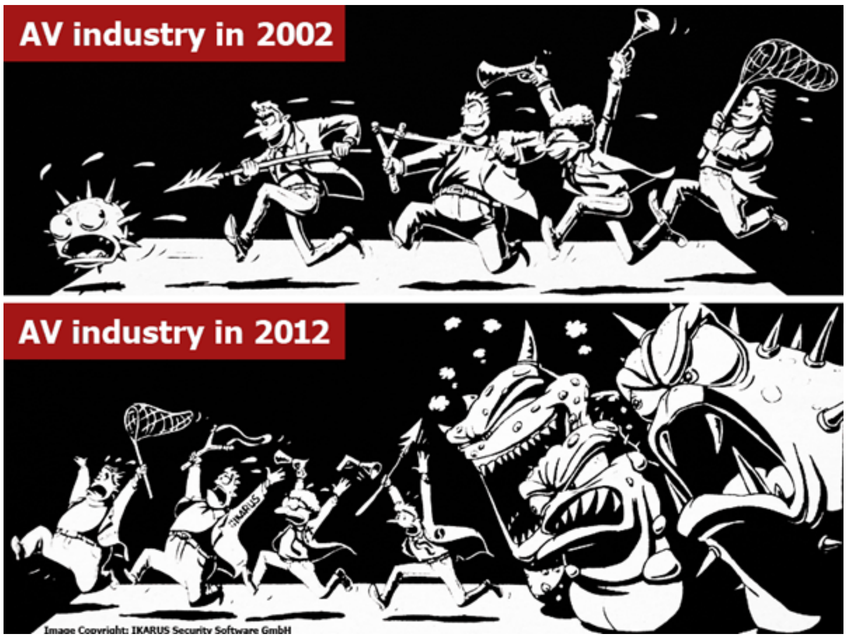


Figure 2.6: Depiction of the perceived malware situation in 2002 and 2012 by the Anti-Virus industry.

This increment in the number of new samples has been, since a while now, unsustainable for malware analysts, so, it is needed an approach capable of automatically detecting malware samples.
Several different attempts have been done in this direction, most of them creating classification models to be applied to testing samples. Recent years approaches have been focusing on dynamic analysis features, due to the high packed samples percentages, however, dynamic analysis has some pitfalls and difficulties that might hinder automatic analysis, as described previously (i.e., anti-debugging and anti-VirtualMachine techniques).
One of the most successful techniques in malware detection is, nowadays, the signature

matching, which is why many automated tools aim at signature generation.
Because of these reasons, this work focuses on static analysis, and, prevalently, on the PE header features, ultimately generating YARA rules.
Ucci et. al [60] present an extensive description of all the methodologies applied and of all the features used in malware related topics. The following sections present a brief description of some of them.
It is worth mentioning that, as stated in [60], the problem of finding a good and standardized samples set is still open, so, it is not yet possible to perform accurate and valuable comparisons of different solutions.

Malwares detection

This set of methodologies aim at determining whether a sample is malicious or not. Examples of malware detection techniques are *signatures* and *classifiers*.

Signature-based approaches. A malware signature is any type of pattern identifying a particular sample or set of samples. Signatures can range from the over-specific executable hash to particular strings and/or opcodes in the executable.
Signatures should be general enough to detect malware variants or entire families, however, they also should be specific enough to find the least false positives possible.

Classification A large number of approaches to the malware detection focused on creating a classifier capable of distinguishing between malicious and benign executables. As an example, some of these methodologies rely on decision trees [55][54], which have been proved effective for malware detection. Moreover, decision trees also have the nice possibility of easily translating the tree to a set of rules, each composed of the AND of each branch from the root to a leaf (i.e., a *rule-based classifier*).
Although classification is a valid approach, it requires an exhaustive goodware and malware samples set. If a malware samples set is fairly easy to get, eventually reducing the set to a particular malware family, it is not that easy to find an acceptable goodware samples set.
Moreover, classification algorithms do not take into account the fast malware evolution pace, which might render useless the model's goodware/malware distinction. On top of that, several studies showed the effectiveness of adversarial Machine Learning in tricking models to mis-classify crafted samples.
Biggio et al. [9] have proved the possibility of crafting PDF malicious samples so to trick Support-Vector-Machines (SVM) and Neural-Networks (NN). Experiments have been made supposing two scenarios: one in which the attacker perfectly knows the victim model and its training dataset, the other in which only the features representations and the type of the model was known. Results show that, given a maximum amount of modifications allowed, SVMs were highly subject to crafting, as well as NNs under particular crafting

procedures (i.e., when considering mimicry methodologies).
Papernot et al. [40], although not targeting specifically the malwares context and not considering modification constraints, have shown that, using the victim model as a black box from which getting classification outputs, they were able to create classifiers (Linear Regression and Deep Neural-Networks) that approximate the victim, from which crafting samples capable of fooling the victim model with high accuracy. Their tests were performed over Amazon Machine Learning service and Google Cloud Prediction having mis-classification rates higher than 84% for both in the most restrictive conditions.

Malware variants and family identification

Some other approaches focused on malware variants (and/or family) detection, for which any tested sample is either associated to its variants (i.e., *variants selection*) or to its families (i.e., *families selection*).
These techniques lie on the idea that malware authors, in order to evade detection, try to generate variants of the same families, by relying on the previously mentioned polimorphic and metamorphic engines. To counter this issue, attempts have been done to associate new malicious executables to their variants or families by applying or removing the most common obfuscation techniques used by polimorphic and metamorphic engines .

Malware Category Detection

This class of methodologies aims at classifying malwares basing on their behaviors and goals. This means that, as opposed to the previous class of approaches, malwares coming from different authors, and not being related (e.g., variants of the same executable) are nevertheless grouped together, provided that they present similar behavioral characteristics.
This type of description can provide useful additional informations to support extensive additional analyses, however, there still isn't a standardized taxonomy of malware categories and this might limit the effectiveness of the approach.

Malware Development Detection

An interesting analysis technique relies on the on-line submissions of samples to scanning services and sandboxes. In particular, malware authors might test their executables on on-line sandboxes and scanning services, before spreading the malware in the wild, to make sure that the executable will not raise any suspect [24][28][5]. Although this approach is fairly new, results in [24] were promising and showed that traces of the development of big malicious campaigns samples where already present in on-line sandboxes before the campaigns began.

Malware Triage

New malwares occur at a constantly growing pace, so, it is usually not possible to take care of all of them in a reasonable amount of time. Because of this, it is needed a fast and fairly reliable prioritization mechanism. Malware triage uses some of the techniques employed in the previous categories to skim low priority samples and make analysts focus on relevant ones, similarly to what happens in an Emergency Room at the hospital.

2.3 Signatures and Automatic Signature Generation Tools

One of the most effective Malware Detection techniques is the signature based one. The term “signature” was introduced during the first years of malwares appearance. Since then, this term has always been indicating a growing number of detection mechanisms.
Initially, signatures were considered as a contiguous sequence of bytes that identify a particular malicious sample, but, as malwares started to evolve, that definition has been extended to comprehend other detection mechanisms [29]. Nowadays, the term signature comprehends any technique capable of identifying (families or classes of) malicious samples. Any Anti-Virus product still relies on a signatures database, that is typically private and based on a custom format. However, lately, there has been a growth in the usage of YARA [68], an open-source tool that supports signature matching and definition.

2.3.1 YARA rule overview

YARA rules are the concise representation of what should be the signature of a malware. Each rule is used by the YARA tool, that compiles it and checks what the rule states on a file, whose name is given as tool argument. A rule follows a C-like syntax and consists of two parts: *strings* and *condition*.
Strings can be hexadecimal sequences, textual strings or regular expressions (eventually, byte sequences can also be expressed through textual strings or regular expressions). The strings typically used in YARA rules represent opcodes[5] of malicious instruction sequences and constant strings (e.g., “/bin/sh”).
The *condition* is a logical expression comprehending (functions of) strings and, eventually, some other operands (such as integer values or modules related operands). An example of a simple YARA rule is in Listing 2.2.

```
rule shell
{
        strings:
          $sh = "/bin/sh"
```

[5]An opcode is the byte sequence that corresponds to a machine instruction (e.g., the instruction “PUSH EAX” corresponds to an opcode of 0x50).

```
        condition:
          $sh
}
```

Listing 2.2: An example of yara rule searching for "/bin/sh" on the entire file.

YARA rules also have the possibility to support for loops and string counts, which allow to create even more complex and targeted rules. For a complete description of all the YARA rules possibilities, it is reminded to the YARA documentation [65].
During the years, and with the growth of the tool's usage, YARA has been extended to support a number of external modules that aid signature generation. These modules allow the researchers to create signatures not only using strings and opcodes, but also relying on more advanced functions.

The "math" module. This module contains an ensemble of mathematical functions that allow to compute statistical measures over the bytes of the analyzed sample. One of the most used functions of this module is the "entropy", which computes the Information Entropy of the bytes of the samples. This particular measure is an heuristic widely used to detect whether a sample is packed or not.

The "hash" module. Another class of supported functions is a subset of all the cryptographic hash functions. Among all the functions, the module supports "MD5","SHA-1","SHA-256" and 32-bit checksums, both of a single string and of an entire portion of the analyzed sample. Although these functions are usually very specific, leading to the match of very targeted byte sequences (which is ultimately the goal of the cryptographic hash algorithms), some of these might be used when creating the hash of only portions of the samples.

The "cuckoo" module. This module introduces an important extension on the YARA rules possibilities. In particular, this module was introduced to support the parsing of any dynamic-analysis report generated by the Cuckoo sandbox. The module allows to access a limited amount of dynamic features, among which: the contacted network hosts, the accessed files and the accessed registry keys.

The "PE" module. The most important module, for the purposes of this work, is the PE one. This module allows to access almost all the fields contained in the header of a Portable Executable. Although access to this type of information can be still performed through the usage of plain rules, the usage of the module dramatically simplifies the process and the readability of the rules.
The power of this module relies also on the almost complete coverage of the header, for which nearly all the fields in the header can be addressed by a rule.

2.3.2 Automatic signature generation tools

As for the time of writing, there are already some automatic signature generation tools, which have also been taken into account as comparative metric for this thesis' framework. These tools are all based on strings and opcodes matching and, the majority of them, produce a set of YARA rules as output.

yarGen

This tool [22] creates YARA rules by searching strings and opcodes in the malicious samples given as input, provided that those are not part of a database of strings and opcodes appearing also on goodwares files. The strings extracted from the samples are also further classified by a Naive-Bayes Classifier so to choose strings that are effectively meaningful, instead of encrypted or compressed sequences.
yarGen also has several possibilities left to the user, such as the usage of strings only or strings and opcodes, or the possibility to use user provided goodwares to extract strings and opcodes that have to be ignored instead of using the predefined database.
In order to create a rule, each string identified by the tool is assigned with a score and only strings having a score high enough will be part of a rule. The scoring system has been hard coded in the tool and assigns points basing on the content of the considered string, eventually adding negative points when the string is also present in the good strings and opcodes database. This scoring system can also be ignored by using the "z0" parameter, which tells the tool to insert in the rules any string that matches the proper requisites, independently on the score they have.
The tool also automatically generates what the author defined *super rules*, that is, rules that can match a sub-set of the malicious samples containing more than one element.

YaraGenerator

This tool [10] works by taking into account strings only in order to generate rules. YaraGenerator is suited not only for PE executables, for which it separately supports imported dlls and functions, but also for e-mail messages, pdfs, Microsoft Office documents and even Javascript or HTML documents.
However, as opposed to the previous tool, it accepts a set of samples and tries to extract features (i.e., strings) common to all of them, which is much more restrictive, and, as a result, it is usually not suited for big sample sets.

Yabin

Yabin [39] is a tool released by the AlienVault Open Threat Exchange (OTX) community to create YARA signatures from the executable code of the malicious samples.
The basic principle of the tool is to find rare functions in any given sample, by looking for

function prologues (e.g., the opcode sequence “55 8B EC”, which is the “PUSH BP; MOV BP, SP” instruction sequence that defines the usual beginning of a function). From all the functions found, the tool then relies on a huge goodwares database to skim common or non-malicious functions.
The main idea behind this approach lies in the *code re-use*: it is not uncommon that malware author recycle code from shared databases when creating new malwares. This would result into having several malwares share part of the code, so, the tool aims at finding those shared bytes.
Although the approach of the tool seems valid, as the authors claim, the tool is designed to work on unpacked samples, because, otherwise, there are good chances that it will identify the packer’s signature.

Icewater

Icewater [43] is not a publicly released tool, however, the author shared some of the rules the tool generated with the community.
The idea behind the tool is quite different from the one of the previous ones, and it has never been disclosed in detail. In general, the tool relies on clustering of malicious samples found on the Internet, from which it is extracted a set of rules that rely on the md5 hash or opcodes of portions of the malicious samples.

BASS

This tool [7] is developed by Cisco-Talos and aims at the automatic generation of ClamAV pattern-based signatures. These signatures are different from the YARA rules, however, the underlying principle is the same: they indicate a sequence of characters or opcodes that uniquely identify malicious samples.
As for the YARA rules, ClamAV signatures support the possibility of using wildcards in the pattern to be matched and to specify the condition as a logical expression of patterns, however, they are more limited in terms of functionalities. For this reason, it is not possible to convert exactly YARA rules to ClamAV signatures, thus, a comparison with the BASS tool is out of the scopes of this work.
BASS works in a more elaborated way than the other tools mentioned. In particular, it relies on already clustered malware samples, from which it generates signatures by finding shared code among the samples.
The shared code is retrieved through a sequence of filtering and elaboration steps, at the end of which a single code sequence, eventually containing wild characters, is returned as signature.
In order to find shared portions of code, BASS disassembles the executable code and finds the most similar functions shared among different samples, by making use of the BinDiff

[6] tool. In particular, BASS searches for the largest connected subgraph of a functions similarities graph generated relying on BinDiff.
From the functions thus extracted, the tool relies on Kam1n0 [59], to filter out library and white-listed functions. Starting from the remaining functions, the tool then extracts, iteratively, by starting from the closest functions couple, the shared Longest-Common-Subsequences (LCS)[7], which, opportunely interleaved with wild characters, will become the samples signatures.
Finally, BASS keeps the generated rules under testing for a limited amount of time, against an internal database of goodwares. If a rule matches a goodware, then the process is started again but, this time, accounting the function that generated the positive match inside the functions' white-list.

[6]BinDiff is a comparison tool for binary files, that allows to find differences and similarities in disassembled code.

[7]A Longest-Common-Subsequence of two strings (in this context, opcodes), is a sequence of *not necessarily contiguous* characters (i.e., bytes), that appear in the same relative order in both strings (e.g., "ACBDA" and "ADDBCDCA" share the longest-common-subsequence "ABDA")

Chapter 3

The proposed framework

The tool presented in this work is inspired by and complementary to the YaYaGen framework, whose goal is the automated YARA rule generation for the Android Operating System. The target of YaYaGenPE is the YARA rule generation for the Microsoft Windows Operating System, which is where most of the business still lies.
Both the tools have been designed keeping in mind the high need for automated and scalable procedures to accurately manage malware detection for their respective target Operating System.
YaYaGen and YaYaGenPE focus on finding, from a given set of samples, a minimum ensemble of features that commonly characterize all of them and, finally, to convert these features into a suitable set of YARA rules.
The result of this behavior is twofold:

- The generated rules do not rely, unless specifically requested to the framework, on a goodwares database. This is, in most of cases, a positive thing, since it is almost impossible to get a comprehensive set of goodwares to use for a correct classification.
- When the set of samples is properly selected, the generated rules are capable of capturing all the relevant features of the malwares and not exclusively those that separate malicious softwares from benign ones.

3.1 Framework's general behavior

Before delving into the details of how the framework actually works and of all its possibilities, it is worth doing a brief description of the overall idea, to make things as clear as possible beforehand.
YaYaGenPE works in 3 steps:

1. **Executable samples features extraction**: in this phase, almost all the fields of the PE header are extracted as features of the executable and, if the user desires to,

it is also possible to include other rules as features of the executable. The provided rules will be converted into their *conjunctive normal form* and split by the "and" conjunction, to create new sub-rules in order to loose some information and make them more general.

2. **Executables clustering:** this step is optional and strictly depends on the algorithm chosen by the user. This phase has been purposely introduced keeping scalability in mind, which is one of the most required features today, given the numbers introduced in chapter 1. Specifically, the clustering allows to manage accurate rules generation for thousands of samples in some hours.

3. **Rule generation**: the final step of the framework generates the YARA rule. Actually, this phase has a broader scope than the one of generating a set of YARA rules, and aims at finding a set of features that is common to many samples given as input. In this phase, there are two possible algorithms the user can rely on (i.e., the greedy algorithm and the clot one), which will be described in detail subsequently.

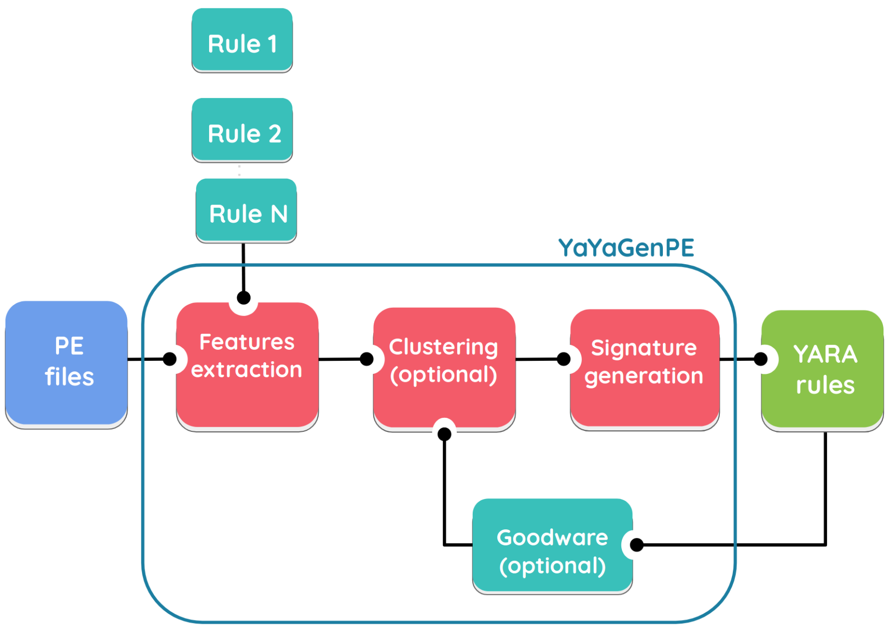


Figure 3.1: Graphical depiction of the entire framework rule generation flow.

3.2 Executable samples feature extraction

In this section it will be described in detail the first step of the tool. As already briefly mentioned, the set of features extracted are basically of two types:

- **PE header fields**: almost all the extractable fields of the executable header are taken.
- **user provided rules**: if any.

3.2.1 PE header fields

An exhaustive description of all the extracted PE header fields has been done in Appendix A, so, in the next lines it will be described only those features that were not explicitly mentioned there.
All the extracted features have been retrieved by making use of the "pefile" [13] Python tool, which, although has a great coverage of the PE header, has some discrepancies with respect to the YARA PE module. This issue implies that, in some particular occasions, it is not possible to successfully extract all the previously mentioned features. To extend the features coverage as much as possible, some of these features are actually obtained by functions purposely written inside the framework, to follow, as much as possible, the YARA PE module extraction procedures, however, even in those cases, there are still some cases in which this might not work.
The main reason behind the choice of these features is that, as introduced in chapter 2, PE header fields, apart from some notable exceptions (i.e., the execution entry point and the Import Address Tables), to the best of this thesis' research background are not reported being tampered by packers.

Imphash feature extraction

The imphash is the hash of the Import Address Table, which keeps track of all the imported functions and the DLLs they belong to. Researchers have found that programs that call the same functions in the same relative order of appearance in the source code will have the exact same Import Address Table. This happens because compilers tend to scan the source code sequentially to identify the requested DLLs of each function and build the Import Address Table accordingly, in a fixed way.
This means that two executables having the same imphash are likely to have been produced by the same source code and the same author, possibly making it a useful feature.
The pseudo-code to extract the imphash from the Import Address Table is described in Listing 3.1, and, although there is an available version on the pefile tool, it has been rewritten to comply with YARA restrictions (which are different from the pefile's ones).

```
imports_string = ""
for dll in pe_object.IAT:
        dll_name = dll.name
    dll_name = remove_suffix(dll_name, ".dll")
    for imported_function in dll.imports:
        if imported_function has Ordinal:
                function_name = convert_ordinal_to_function_name(Ordinal)
        else:
                function_name = imported_function.Name
        imports_string += ("," + dll_name+ "." +function_name)
return md5(imports_string)
```

Listing 3.1: Imphash extraction pseudo-code. The description follows the original Python code as much as possible, with some modifications to help comprehension.

Overlay data extraction

The Overlay data is a section of bytes that are not addressed by any field of the PE header. As a consequence, this data is not loaded automatically by the loader when creating the executable image in memory. However, the Overlay section might contain malware payloads, together with some other informations, not necessarily malicious.
Overlay data lie at the end of the executable, but it is not always straightforward to trace its beginning, especially when dealing with tampered PE headers (which is usually the case for malicious samples). Nevertheless, the PE module of YARA allows to address both the size and the offset of an eventual Overlay section in the executable.
At the time of writing, YaYaGenPE uses the function designed in the pefile tool to extract the Overlay section, but, as for the Imphash, this feature is not always coherent with the one required by YARA, so, if the two do not match, this feature will be ignored.

3.2.2 User provided rules features

In order to improve the rules efficacy, YaYaGenPE offers the possibility to use, as features of a malicious sample, the fact that it matches other YARA rules, that are retrieved by simplifying user provided YARA rules.
The reasons behind this new set of features are multiple:

- Several malware authors tend to reuse code or functions from other malicious samples already in the wild, either because the author was the developer of the other samples or because he/she relies on a commonly shared database of functions. As a consequence, using previously matching rules (or part of them) might help identifying malicious samples or even families.
- Usually, rules that have been written by domain experts are extremely effective and well targeted, as opposed to automatically generated rules. However, it might happen that those rules are over-specific for the new context in which they are currently being applied, so, they have to be accurately simplified.
- YaYaGenPE exclusively focuses on the PE header field, so, it does not take into account the source code of the executable. Adding opcodes to the rule generation

might overcome this limitation, however, several rules shared with the community already contain opcodes targeted for particular malwares. For this reason, it might be reasonable to try to use those rules, instead of finding meaningful opcodes in the targeted malwares.

Before using user provided rules as features, it is necessary to make them more general, since many of them, especially those written by domain experts, tend to be tailored for a given set of malwares.

Rules normalization and simplification

In order to simplify the rules, the idea of this work was that of converting the rules' conditions into their *conjunctive normal form* and then splitting each of these normalized conditions into a list of conditions. Before explaining the reasons behind this choice and eventual other options, it is useful to give a definition of the terms that will be used in the next lines.

Definition 3.2.1. In mathematical logic, a **literal** is an atomic formula or its negation.

Definition 3.2.2. An atomic formula is any formula that does not contain any logical connective.

The definition of formula strictly depends on the type of logic that is being used. In propositional logic, a formula can be:

- a propositional variable, such as True, False or any named variable (e.g., A);
- the negation (NOT) of a formula (e.g., $\neg A$);
- the concatenation of two formulas, through a binary connective, that is one of AND ($\wedge$), OR ($\vee$), implication ($\implies$), bi-implication ($\iff$).

Although YARA rules are more complex than plain propositional logic, in that they contain functions, that are typical of predicative logic, in this work they have been approximated to propositional logic conditions, since it is accurate enough for the applicative context.

Definition 3.2.3. In Boolean logic, a formula is in **Conjunctive Normal Form (CNF)** if it is a conjunction (AND) of one or more clauses, where a clause is a disjunction (OR) of literals.

An example of a CNF formula might be:

$$(A \vee B \vee \neg C) \ \wedge \ (\neg B \vee C),$$

where the two parts separated by the AND operator (i.e., $(A \vee B \vee \neg C)$ and $(\neg B \vee C)$) are the clauses.

Now that the basic concepts have been introduced, it is possible to explain why this simplification process is performed. Specifically, if the previous CNF formula would represent a YARA rule condition, for the rule to match, both the clauses have to be true. However, it might happen that, for over-specific rules, one of the clauses is general enough to be adopted for other malicious samples, while the other might not. For this reason, when the user provides some rules, the tool does not use the entire condition as feature, but converts it into CNF and uses all the resulting clauses as new conditions.
It is however worth mentioning that, if a rule is already general enough, the splitting of its CNF will not alter the final result, since the rules produced by the tool will be the AND of all the features found by the chosen algorithm. This means that, if all clauses of a rule's CNF are general enough to be selected, in the final rule there will be, together with other features, all those clauses, concatenated by AND conjunctions (i.e., the original rule).

Conversion of YARA rule condition to logical proposition. The algorithm to determine the conjunctive normal form from any given propositional logic formula is listed in Algorithm 1.

Algorithm 1 Conversion of any propositional logic formula to its conjunctive normal form (CNF)

```
1: Convert each implication $P \implies Q$ to its equivalent form $\neg P \lor Q$
2: Convert each bi-implication $P \iff Q$ to its equivalent form $(\neg P \lor Q) \land (P \lor \neg Q)$
3: Repeatedly replace $\neg(P \land Q)$ with $\neg P \lor \neg Q$
4: Repeatedly replace $\neg(P \lor Q)$ with $\neg P \land \neg Q$
5: Repeatedly replace $\neg(\neg P)$ with P.
6: Repeatedly replace $P \lor (Q \land R)$ with $(P \lor Q) \land (P \lor R)$
```

In order to apply the mentioned algorithm, it is however necessary to convert (and approximate) each rule condition to a propositional formula. This implies making some assumptions and simplifications over which constructs to convert to propositional variables. As for the current version of the tool, the following substitutions are made:

1. any "for" construct is converted to a single propositional variable, even if it contains nested "for" constructs. As an example, the construct:

$$for\ i\ in\ (0..100) : (for\ j\ in\ (300..500) :\ \$a\ at\ i\ and\ \$b\ at\ j)$$

   becomes a single variable (e.g., the variable "A").

2. any construct like "all of ...", "any of ..." or "n of ...", where n is an integer, is converted to a single variable (e.g., "B").

3. any other remaining logical construct that does not contain a connective is converted to a single variable.

Creating sub-rules from the YARA rule condition. Once the YARA rule condition has been converted into a propositional logic formula, Algorithm 1 is applied to identify the CNF of the condition.
Finally, each *clause* (OR formula in between two ANDs) is selected and its original YARA rule constructs are restored. Each clause identified will thus become a new YARA rule. When converting clauses' variables back to their original value, it is also checked whether the original construct is in a list of too generic conditions or not. If it is, then the construct is omitted from the final sub-rule. This final step is fundamental to remove conditions that will, if present, make the sub-rule match for almost every sample.

Putting things together: complete conversion example

As an example, lets consider the YARA rule in Listing 3.2, that matches a sample of one of the most famous cyber-physical malware campaigns.

```
rule Stuxnet_Malware_3 {

    meta:
        description = "Stuxnet␣Sample␣-␣file␣~WTR4141.tmp"
        author = "Florian␣Roth"
        reference = "Internal␣Research"

    strings:
        $x1 = "SHELL32.DLL.ASLR." fullword wide
        $s1 = "~WTR4141.tmp" fullword wide
        $s2 = "~WTR4132.tmp" fullword wide
        $s3 = "totalcmd.exe" fullword wide
        $s4 = "wincmd.exe" fullword wide
        $s5 = "http://www.realtek.com0" fullword ascii
        $s6 = "{%08x-%08x-%08x-%08x}" fullword wide

    condition:
        (uint16(0) == 0x5a4d and filesize < 150KB and ($x1 or 3 of ($s*)))
            or (5 of them)
}
```

Listing 3.2: Example of Yara rule matching the Stuxnet malware.

Following the previous conversion steps, these substitutions will be defined:

- "uint16(0) == 0x5a4d" becomes variable "A";
- "filesize < 150KB" becomes variable "B";
- "$x1" becomes variable "C";
- "3 of ($s*)" becomes variable "D";

- “5 of them” becomes variable “E”;

the condition becomes something like:

$$(A \text{ and } B \text{ and } (C \text{ or } D)) \text{ or } E.$$

The previous is a plain propositional logic formula, for which it is possible to apply Algorithm 1. The result of the application of the algorithm is:

$$(A \text{ or } E) \text{ and } (B \text{ or } E) \text{ and } (C \text{ or } D \text{ or } E).$$

Finally, selecting each clause and reverting back each variable to the original construct, the rules in Listing 3.3 are generated.

```
rule Stuxnet_Malware_3_0 {

    strings:
        $x1 = "SHELL32.DLL.ASLR." fullword wide
        $s1 = "~WTR4141.tmp" fullword wide
        $s2 = "~WTR4132.tmp" fullword wide
        $s3 = "totalcmd.exe" fullword wide
        $s4 = "wincmd.exe" fullword wide
        $s5 = "http://www.realtek.com0" fullword ascii
        $s6 = "{%08x-%08x-%08x-%08x}" fullword wide

    condition:
        uint16(0) == 0x5a4d or (5 of them)
}

rule Stuxnet_Malware_3_1 {

    strings:
        $x1 = "SHELL32.DLL.ASLR." fullword wide
        $s1 = "~WTR4141.tmp" fullword wide
        $s2 = "~WTR4132.tmp" fullword wide
        $s3 = "totalcmd.exe" fullword wide
        $s4 = "wincmd.exe" fullword wide
        $s5 = "http://www.realtek.com0" fullword ascii
        $s6 = "{%08x-%08x-%08x-%08x}" fullword wide

    condition:
        filesize < 150KB or (5 of them)
}

rule Stuxnet_Malware_3_2 {

    strings:
        $x1 = "SHELL32.DLL.ASLR." fullword wide
        $s1 = "~WTR4141.tmp" fullword wide
        $s2 = "~WTR4132.tmp" fullword wide
        $s3 = "totalcmd.exe" fullword wide
        $s4 = "wincmd.exe" fullword wide
        $s5 = "http://www.realtek.com0" fullword ascii
        $s6 = "{%08x-%08x-%08x-%08x}" fullword wide

    condition:
        $x1 or 3 of ($s*) or (5 of them)
}
```

Listing 3.3: Result of the sub-rules generation.

Results in Listing 3.3, although correct, show the reason behind the last point of the previous paragraph (i.e., filtering out too much generic literals). In particular, "uint16(0) == 0x5a4d" will always match a PE executable (referring to Appendix A) and "filesize < 150KB" might as well match a big number of executables.
Because of this reason, once defined the clauses and converted them back, too general literals are removed from the newly created rules. As a result, the first two rules will become as shown in Listing 3.4.

```
rule Stuxnet_Malware_3_0 {

    strings:
        $x1 = "SHELL32.DLL.ASLR." fullword wide
        $s1 = "~WTR4141.tmp" fullword wide
        $s2 = "~WTR4132.tmp" fullword wide
        $s3 = "totalcmd.exe" fullword wide
        $s4 = "wincmd.exe" fullword wide
        $s5 = "http://www.realtek.com0" fullword ascii
        $s6 = "{%08x-%08x-%08x-%08x}" fullword wide

    condition:
        5 of them
}
```

Listing 3.4: First two rules of Listing 3.3 after cleaning the generic constructs.

The two resulting rules will be identical, so, only one of them is preserved, while the other is removed from the generated sub-rules.

Rules to features conversion. The previous steps describe the process of simplification of the user provided rules. However, these rules are *not yet* the malicious samples features. In order to create the proper features from these rules, it is, indeed, tested each rule against each input malware and, when a rule matches, it is added to the remaining set of features associated to the matched sample.

3.3 Executables clustering

This step of the tool is optional and performed only when requested by the user, although it is suggested to request it when the size of the samples set is fairly high (clustering should be adequate for samples sets of 10 elements or more).
The reason behind the usage of clustering has to be found on the balance between rules outcomes and computational complexity and time required by the two basic algorithms (i.e., greedy and clot). As will be explained in the next section, the greedy algorithm is quite fast, however, when the samples set size increases significantly, it tends to produce rules with a small amount of literals. This might mean that rules are too generic and, consequently, they can generate a relatively high number of false positives. On the contrary, the clot algorithm tends to be more accurate than the greedy one, but its computational time is higher.
In order to account for both, clustering of executable samples seems to be the most reasonable solution.
There are lots of clustering algorithms that have been developed and studied during the years, however, some require to set up the number of clusters the user wants (e.g., the K-Means algorithm or any hierarchical clustering algorithm, if there is no alternative choice for the dendrogram's cutting point). This parameter is not determinable a priori in the context of malware clustering, especially since the tool does not know the samples it will be provided with until runtime.
For this reason, the choice of the clustering algorithm has been restricted to those that are capable of automatically defining the number of clusters produced.
The algorithms selected for the tool are of two different clustering classes:

- **density-based clustering**: this class of algorithms generates clusters by identifying, in the areas of the features space, points connected together by an appropriate level of density. The most famous algorithm of this class is DBSCAN, whose acronym stands for *density-based spatial clustering of applications with noise.*

- **monothetic divisive clustering**: this class of algorithms is not among the most famous ones, but, for the purposes of this work it might be meaningful to give it a try. In particular, a monothetic divisive (hierarchical) clustering algorithm, as the name says, performs an hierarchical divisive clustering by taking into account *one* feature at each split. This approach allows to read the dendrogram thus created as if it was a decision tree.

Before delving into the details of the mentioned approaches, it is necessary to talk a little about used metrics and features' domains.

3.3.1 Features' domains and metrics

As for all the data analysis problems, the first step to take, before choosing the algorithms, is to decide which will be the numerical domain of each feature and, once defined, the most appropriate distance for that domain.

Features' domain choice

During the development of the tool, it has been chosen to consider all the features as binary, eventually converting features relying on strings and integers to binary ones.
The reasons behind this choice are multiple:

- most of the features of the PE header are binary ones, such as those involving flags of any field, Imported functions and DLLs or even the rules matching, if the user provides them.
- features relying on integer values, which are almost all the remaining ones, tend to assume discrete values and, in almost every case, are not that meaningful for the identification of the clusters. As an example, features like PE header's image base field, section alignment, disk alignment or sections starting addresses usually tend to be always multiple of a page or of a disk section (in the case of disk alignment), and this holds even for executables completely different from each other. On the contrary, resources offsets and even size are really mutable but the same values of these fields might be valuable for identifying variants of the same samples.
- features containing string values have no clear distance definition, apart from the equality concept. Because of this, it is appropriate to convert them to binary, even if it adds features redundancy.

Although some features selection might be done just before applying density-based clustering, one might ask why aren't the features containing strings and integers always removed. The reason is twofold:

- as explained in chapter 2, malwares tend to rely either on polimorphic/metamorphic engines or on packers. Although these tools modify the content of the executable, they usually keep untouched most of the integer values in the PE header, as they usually tamper the Import Table and the executable's entry point.
- when packers modify section names, which is one of the few string features, they tend to give, in some cases, names that are related to the packer in some way or unconventional ones. In these cases, it might be helpful to keep this information for the clustering procedure.

Distance choice

When choosing a distance definition, it is necessary to take into account the mathematical domain of the features, and the meaning the distance has with respect to the applicative field.
Because of the choice made on the previous lines, the selection of the distance has to be done among those compatible with binary values. An exhaustive listing of similarities and distances for binary features is in [18].
The similarities (and, consequently, the distances) that have been chosen for the tool were the Jaccard and the Russell-Rao ones. However, before describing the chosen similarities, it is worth defining some quantities, which will be referred later.
In the context of binary features, each data sample (in this case, an executable) can be represented as a binary array of length equal to the total number of distinct features. Each binary value is then set to 1 if the corresponding feature is present in the sample or 0 otherwise.
Given a set of N features, and two samples p and q, it is possible to define 4 quantities:

$$a = \sum_{i=1}^{N} p_i q_i, \tag{3.1a}$$

$$b = \sum_{\substack{i:p_i=0 \\ \wedge \\ q_i\neq 0}} 1, \tag{3.1b}$$

$$c = \sum_{\substack{i:p_i\neq 0 \\ \wedge \\ q_i=0}} 1, \tag{3.1c}$$

$$d = \sum_{\substack{i:p_i=0 \\ \wedge \\ q_i=0}} 1. \tag{3.1d}$$

The sum of all these quantities gives the total number of features (N).
These quantities respectively represent: the number of features p and q have in common (a), the number of features p has that q doesn't (b), the number of features q has that p doesn't (c) and the number of features both p and q do not have (d).

Jaccard Similarity. The Jaccard similarity is a statistical measure that evaluates how close two samples are by measuring the ratio between the size of the common features and the size of the union of the features present in at least a sample.
Mathematically, given two features sets A and B, coming from two different samples, the Jaccard similarity is defined as:

$$J(A, B) = \frac{card(A \cap B)}{card(A \cup B)}.$$

Equivalently, using the quantities defined in Equation 3.1:

$$J(p,q) = \frac{a}{a+b+c}$$

Russell Rao similarity. This similarity is less common than the previous one, however, it has the advantage of considering also the absence of features. Specifically, the Russell Rao similarity of two samples is defined as the ratio between the total number of common features and the total number of features in the domain (i.e., the length of the array representing all the features).
Referring back to Equation 3.1, the Russell Rao similarity can be expressed as:

$$R(p,q) = \frac{a}{a+b+c+d}$$

Jaccard and Russell Rao differences. From the previous definitions, it is easy to see that the Jaccard similarity tends to evaluate the closeness of two samples *relative* to the features possessed by at least one of them. On the contrary, the Russell Rao similarity measures how similar two samples are in an absolute way, that is, by considering all the possible features the samples might have.
To better explain the difference, and why both similarities are valuable in the context of this work, lets assume to have three arrays of features: A = 1101011011, B = 1100000000 and C = 0101000000.
The Jaccard similarities for each couple of arrays are:

$$J(A,B) = \frac{2}{7},\ J(B,C) = \frac{1}{3} \text{ and } J(A,C) = \frac{2}{7}$$

As a result, the closest couple, considering the Jaccard similarity, is (B,C), even though the two of them have only one common component, while (A,B) and (A,C) share two features each. When computing the Russell Rao similarity, instead:

$$R(A,B) = \frac{2}{10},\ R(B,C) = \frac{1}{10} \text{ and } R(A,C) = \frac{2}{10}$$

This means that, when using the Russell Rao similarity, couples (A,B) and (A,C) are equivalently similar and have higher similarity than the (B,C) couple.
In the context of features clustering for rule generation, although the Jaccard similarity is a valid measure, it might be useful to take into account the Russell Rao one, since it tends to privilege elements that have many features in common. Having an high number of common features in a cluster is desirable in the context of rules generation since it helps creating detailed rules, which might be less prone to false positives.

3.3.2 Density-based clustering algorithms

Many clustering algorithms work under the assumption that data points are distributed following a mixture of several Gaussian distributions. This assumption leads to the creation of spherical clusters, which, in some cases, might not be adequate. Moreover, some of those algorithms require to set the number of clusters to be determined, which, is not easy to define a-priori and it is typically dependent on the instance of the problem.
Density-based algorithms, on the contrary, do not make assumptions about data distribution, and try to discover arbitrarily shaped clusters (Figure 3.2). The principle behind density based clustering is to find connected, dense areas in the data (features) space, separated by sparser areas [21]. Consequently, the number of clusters the algorithm generates strictly depends on the number of connected dense areas found.

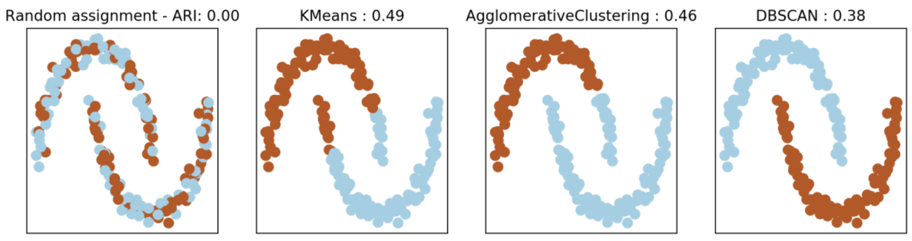


Figure 3.2: Comparison of different clustering algorithms when dealing with non-rounded shape clusters.

The results of the algorithm are dependent on the definition of “dense areas” and of “connected” points. Different definitions of the previous terms define different density-based clustering algorithms.

3.3.3 HDBSCAN clustering algorithm

The HDBSCAN algorithm is an improvement of the density-based clustering procedure. It tries to overcome all the limitations that density-based clustering algorithms have [11]. In practice, this is done by adding hierarchical clustering on top of a DBSCAN variant [11][12][36], by varying the density parameter for all the possible values it can have and keeping track of any change in the clusters when doing so.

DBSCAN algorithm

DBSCAN works on the concepts of “dense” areas. In order to define when an area is “dense”, it is thus needed to define a minimum density threshold, and a common unit area where to compute the density. Specifically, DBSCAN requires that the user provides a minimum number of points (data samples) *MinPts* as density threshold and a radius ϵ

from which it is created a circular area in which density is evaluated.
Using the previous two parameters, it is possible to define the concepts of *core points* and of *density-reachability*.

Definition 3.3.1. A data sample is a core point if the neighborhood of radius ϵ of the sample contains at least *MinPts* points.

Definition 3.3.2. A point q is directly density-reachable from a point p if p is a *core* point and $distance(p,q) < \epsilon$.

Definition 3.3.3. A point q is (generally) density-reachable from a point p if there is a path $p_1, \dots, p_N$ such that $p_1 = p$, $p_N = q$ and, for each i in $1, \dots, N-1$, p_{i+1} is *directly density-reachable* from p_i.

From these definitions, it is possible to classify each point of the dataset as:

- *core point.*
- *border point*: a data sample is a border point if the neighborhood of radius ϵ of the sample contains less than *MinPts* points and at least one of them is a *core* point.
- *noise point*: a data sample is a noise point if the neighborhood of radius ϵ of the sample contains less than *MinPts* points and none of them is a *core* point.

The algorithm starts from a point of the dataset and classifies all the points according to one of the previous definitions. While doing so, the algorithm simultaneously defines the clusters, by grouping together all the points that are density-reachable from at least another point (i.e., the *core* and *border* points).
The algorithm's pseudocode is in Algorithm 2.
Although DBSCAN works well for data spaces in which clusters are not rounded shaped, there are some limitations of this approach. The first of these is the difficulty in establishing which are the most appropriate threshold and ϵ parameters. Even more relevant, this algorithm can provide good results only for data spaces where density is sufficiently homogeneous. Indeed, when the algorithm is provided with a dataset with different density areas, results are not as good as expected.
HDBSCAN tries to overcome both these limitations at the cost of a slower procedure.

DBSCAN* algorithm

This algorithm is a variant of the standard DBSCAN that removes the concept of border points. With this restriction in mind, it is possible to define a concept similar to the *direct density-reachability*, that is ϵ-reachability. In particular, given the previously introduced parameters *MinPts* and ϵ, the ϵ-reachability is defined as follows:

Definition 3.3.4. Two points p and q are ϵ-reachable if they are *core* points and $distance(p,q) \leq \epsilon$.

Algorithm 2 DBSCAN pseudo-code.

```
$C \leftarrow 0$
for Point p in database DB do
    if $label(p) \neq undefined$ then
        continue
    end if
    $Neighbors = get\_Neighbors(DB, distance, p, eps)$
    if $|Neighbors| < minPts$ then
        $label(p) \leftarrow "Noise"$
        continue
    else
        $C \leftarrow C + 1$
        for q in Neighbors do
            if $label(q) == "Noise"$ then
                $label(q) \leftarrow C$
            end if
            if $label(q) \neq undefined$ then
                continue
            end if
            $label(q) \leftarrow C$
            $Neighbors\_q = get\_Neighbors(DB, distance, q, eps)$
            if $|Neighbors\_q| \geq minPts$ then
                $Neighbors \leftarrow Neighbors \cup Neighbors\_q$
            end if
        end for
    end if
end for
```

From this definition, it follows:

Definition 3.3.5. Two points can be defined as *density-connected* if they are directly or transitively ϵ-reachable.

The DBSCAN* algorithm differs from the DBSCAN in the way it defines the clusters. In particular, the algorithm defines each cluster C as the set of points of the dataset that are *density-connected.*

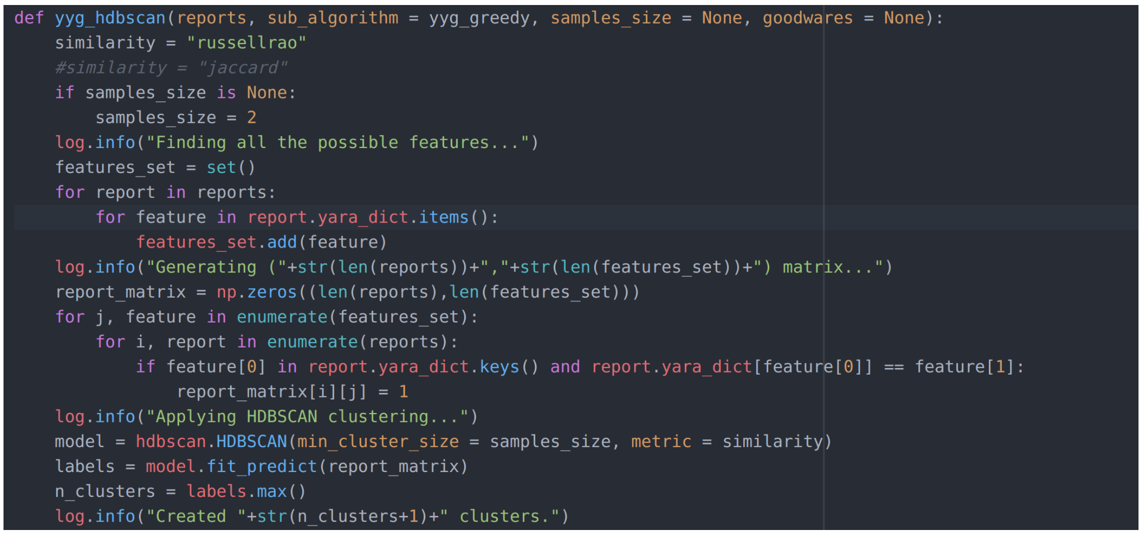
HDBSCAN algorithm

The idea behind HDBSCAN is to extend DBSCAN* to support an hierarchy of DBSCAN* clustering results for each possible ϵ value. There are different interpretations of the algorithm's procedure, which are exhaustively described in [36].
Although the explanation of the algorithm is more elaborated than the following one,

and not strictly linked to the DBSCAN* algorithm, it is possible to conceptually interpret HDBSCAN as a way to find clusters that persist for several ϵ values when running DBSCAN* over all the possible ϵ. This clusters' selection removes the choice of the ϵ parameter and allows to select clusters properly even when dealing with varying density datasets.
Moreover, if the persistence score of each cluster is defined properly, the problem of finding a set of non-overlapping clusters that maximizes the total persistence can be seen as a constrained optimization problem and it has a straightforward solution.

Applying the HDBSCAN algorithm to the dataset

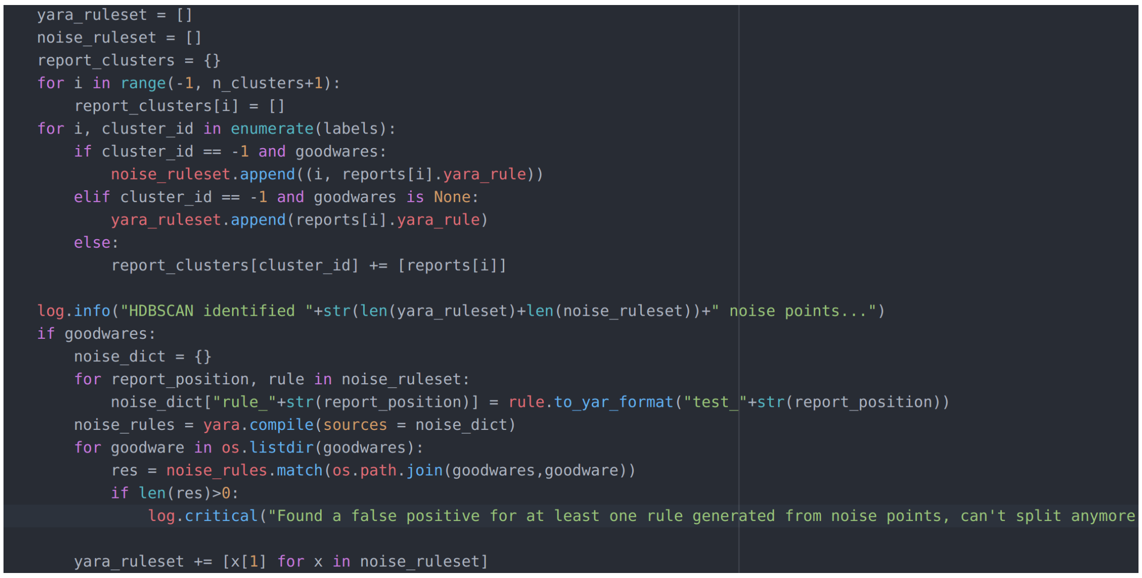
This work uses the implementation of the HDBSCAN algorithm in [58]. It requires, as input parameter, a matrix of N rows and M columns, where N is the total number of samples and M is the total number of features, and a metric. The matrix contains, at each entry (n,m) a 1 if the n-th sample has the m-th feature, or 0 otherwise. The metrics chosen, as mentioned previously, were the Jaccard and the Russell-Rao ones, both natively supported by the library (Figure 3.3).



```python
def yyg_hdbscan(reports, sub_algorithm = yyg_greedy, samples_size = None, goodwares = None):
    similarity = "russellrao"
    #similarity = "jaccard"
    if samples_size is None:
        samples_size = 2
    log.info("Finding all the possible features...")
    features_set = set()
    for report in reports:
        for feature in report.yara_dict.items():
            features_set.add(feature)
    log.info("Generating ("+str(len(reports))+","+str(len(features_set))+") matrix...")
    report_matrix = np.zeros((len(reports),len(features_set)))
    for j, feature in enumerate(features_set):
        for i, report in enumerate(reports):
            if feature[0] in report.yara_dict.keys() and report.yara_dict[feature[0]] == feature[1]:
                report_matrix[i][j] = 1
    log.info("Applying HDBSCAN clustering...")
    model = hdbscan.HDBSCAN(min_cluster_size = samples_size, metric = similarity)
    labels = model.fit_predict(report_matrix)
    n_clusters = labels.max()
    log.info("Created "+str(n_clusters+1)+" clusters.")
```

Figure 3.3: Creation of the NxM matrix from the reports given as function parameter.

At the end of the clustering procedure, the HDBSCAN algorithm will return a set of clusters and a set of noise points. For each cluster, it is applied the rules generation algorithm. However, it is necessary to decide how to act when dealing with the noise points' rules generation. The current choice is to create a new rule for each noise point, as specific as possible(Figure 3.4). This approach generates rules that might be over-specific, however, grouping together all the noise points in a single cluster and applying the rules generation algorithm on it have shown, experimentally, poor results.



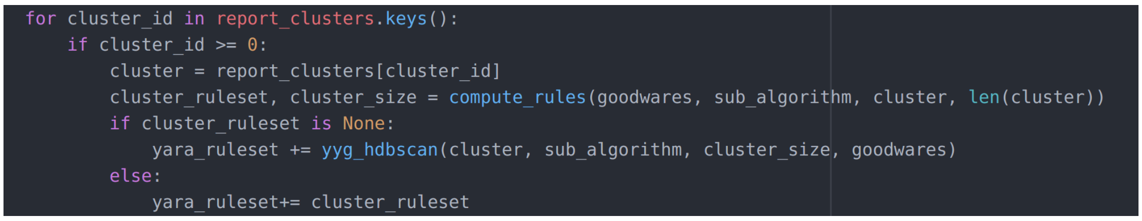
```
yara_ruleset = []
noise_ruleset = []
report_clusters = {}
for i in range(-1, n_clusters+1):
    report_clusters[i] = []
for i, cluster_id in enumerate(labels):
    if cluster_id == -1 and goodwares:
        noise_ruleset.append((i, reports[i].yara_rule))
    elif cluster_id == -1 and goodwares is None:
        yara_ruleset.append(reports[i].yara_rule)
    else:
        report_clusters[cluster_id] += [reports[i]]

log.info("HDBSCAN identified "+str(len(yara_ruleset)+len(noise_ruleset))+" noise points...")
if goodwares:
    noise_dict = {}
    for report_position, rule in noise_ruleset:
        noise_dict["rule_"+str(report_position)] = rule.to_yar_format("test_"+str(report_position))
    noise_rules = yara.compile(sources = noise_dict)
    for goodware in os.listdir(goodwares):
        res = noise_rules.match(os.path.join(goodwares,goodware))
        if len(res)>0:
            log.critical("Found a false positive for at least one rule generated from noise points, can't split anymore

    yara_ruleset += [x[1] for x in noise_ruleset]
```

Figure 3.4: Creation of one rule for each noise point identified and clusters grouping. The rule creation step is just the conversion of the noise point's features to a YARA rule composed as the logical conjunction of all the features.

As shown in Figure 3.4, with this last choice, if the user provides some goodwares, there is no way to avoid false positives, if there is any for the noise points' rules. Nevertheless, if any noise point generated a rule detecting at least a false positive, that would be always present, no matter the cluster the noise point is in. Specifically, if a noise point rule detected a false positive, it would mean that all the noise sample's features are not enough to distinguish it from the detected false positive.
The remaining samples (i.e., those that are not noise points) are grouped in their respective clusters (Figure 3.4). For each cluster it is then applied the procedure of rules generation (Figure 3.5).



```
for cluster_id in report_clusters.keys():
    if cluster_id >= 0:
        cluster = report_clusters[cluster_id]
        cluster_ruleset, cluster_size = compute_rules(goodwares, sub_algorithm, cluster, len(cluster))
        if cluster_ruleset is None:
            yara_ruleset += yyg_hdbscan(cluster, sub_algorithm, cluster_size, goodwares)
        else:
            yara_ruleset+= cluster_ruleset
```

Figure 3.5: Creation of the rules for each cluster identified (i.e., those whose samples' cluster_id is at least zero). The rule creation step is performed by the "compute_rules" function.

As shown in Figure 3.5, if the user provides some goodwares, and any of these produces a false positive for a cluster, the set of rules returned by the "compute_rules" function will

be empty, and, in that case, the HDBSCAN algorithm is recursively applied only to that clusters' points, so to identify more specific sub-clusters and, consequently, more specific rules.
Finally, once each sample is covered by at least a rule, the total rule set is returned.

3.3.4 Monothetic clustering algorithms

The monothetic clustering algorithms are a set of procedures that define clusters whose components have some features in common, instead of being generally "close" to each other. Specifically, a cluster is called *monothetic* if a conjunction of logical properties is both necessary and sufficient for the membership in the cluster [15]. In practice, this is done, in the case of hierarchical clustering, by splitting (or merging) data objects using a single variable at a time.
This type of clustering methodologies is not widespread among the community, however, the previous definition is what is desirable for the rules generation procedure because it allows, while creating the clusters, to identify some of the features that will be included in the final rules.
For the purposes of this work, the monothetic algorithms that have been taken into account are those involving a hierarchical divisive clustering, and, in particular, they are the "Unsupervised Decision Tree" [6] and the "DIVCLUS-T" [16].
These two algorithms are conceptually similar each other, however, they require that the user defines a minimum number of samples per leaf node or the number of desired clusters (these two are the requirements for the "Unsupervised Decision Tree") or the dendrogram's cutting point (for the DIVCLUS-T).
These requirements, as already mentioned, are not easily determinable in an automated context, so, the algorithms have been slightly adapted to stop when an homogeneity criterion is matched.
Although it is well known that developing ad-hoc clustering algorithms might not be the best choice, the applicative context and the clusters' subsequent usage for rule generation justify the introduction of these variations from the standard procedures.
Through all the entire work, this clustering variant will be referred to as "Unsupervised Decision Tree"(UDT), which, although being also the name of one of the two clustering procedures it was inspired by, is auto-explicative of the clustering outcome.

3.3.5 Building the clusters tree

The generation of the clustering tree in the tool follows the same structure of the ones in [6] and [16], however, it differs for the evaluation of the quality of the split and for the stopping criterion, which are linked together.
The algorithm consists of a recursive procedure that works on a sub-partition of the initial dataset. In the following description, it will be considered the recursive function over a

generic partition of the dataset, which, at the beginning, corresponds to the entire dataset.

Features extraction and features selection

The procedure starts by counting, for each feature present in at least a data sample of the considered partition, the number of occurrences of the feature. Among these features, only those that are shared by at least a minimum percentage of data samples m and by at most $(1-m) \cdot N$ samples, with N total number of samples in the partition, are preserved. The reason behind this features selection process is twofold:

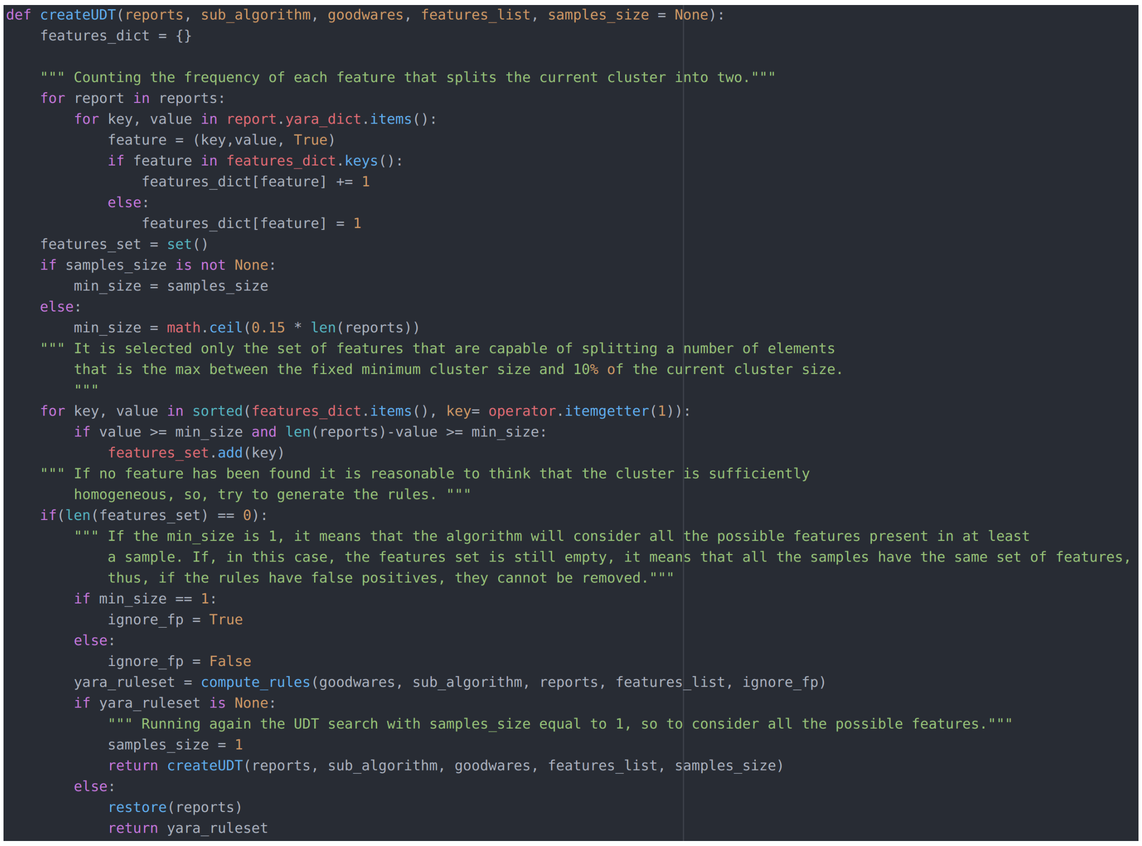
- considering all the possible features would slow down the clustering procedure too much, especially when working on big datasets at the beginning of the procedure;
- features that are rare in the partition, although valuable, might create too small clusters, and, considering the subsequent rule generation procedure inside the final cluster, this might lead to over-specific rules.

It is worth mentioning that, as the recursion goes deeper, and partitions get smaller, the features selection becomes lesser and lesser selective, allowing the algorithm to be sufficiently generic at the beginning, while more specific towards the end.
Due to this features selection, it is possible that the set of selected features will be empty, because all the features are either too much rare or too much frequent in the partition. In that case, the clustering algorithm stops and the rules generation procedure starts, as shown in Figure 3.6.
Similarly to HDBSCAN, if the rules generation procedure creates rules that match any of the user provided goodwares, the rule set will be empty, so, the procedure will be repeated from scratch, this time considering all the possible features (i.e., $m = \frac{1}{N}$).
In this last case, if the features set is still empty, it means that all the samples share the same set of features, so, if the rules generated from here will produce false positives, they cannot be extended any further and, as a consequence, they will still produce false positives.

Selecting the best splitting feature

Once all the valid features have been selected, it is necessary to define which of these is the best one. Algorithms in [6] and [16] propose two different approaches for the choice of the best feature.
In particular, DIVCLUS-T [16] proposes the usage of the inertia criterion for the considered partition, which is a generalization of the sum of square errors criterion. This criterion, applied to a bi-partition results in the maximization of the between-cluster inertia of the partition, which is:

$$B(A_l, A_{\bar{l}}) = \frac{\mu(A_l)\mu(A_{\bar{l}})}{\mu(A_l) + \mu(A_{\bar{l}})} distance^2(g(A_l), g(A_{\bar{l}}))$$



```python
def createUDT(reports, sub_algorithm, goodwares, features_list, samples_size = None):
    features_dict = {}

    """ Counting the frequency of each feature that splits the current cluster into two."""
    for report in reports:
        for key, value in report.yara_dict.items():
            feature = (key,value, True)
            if feature in features_dict.keys():
                features_dict[feature] += 1
            else:
                features_dict[feature] = 1
    features_set = set()
    if samples_size is not None:
        min_size = samples_size
    else:
        min_size = math.ceil(0.15 * len(reports))
    """ It is selected only the set of features that are capable of splitting a number of elements
        that is the max between the fixed minimum cluster size and 10% of the current cluster size.
        """
    for key, value in sorted(features_dict.items(), key= operator.itemgetter(1)):
        if value >= min_size and len(reports)-value >= min_size:
            features_set.add(key)
    """ If no feature has been found it is reasonable to think that the cluster is sufficiently
        homogeneous, so, try to generate the rules. """
    if(len(features_set) == 0):
        """ If the min_size is 1, it means that the algorithm will consider all the possible features present in at least
            a sample. If, in this case, the features set is still empty, it means that all the samples have the same set of features,
            thus, if the rules have false positives, they cannot be removed."""
        if min_size == 1:
            ignore_fp = True
        else:
            ignore_fp = False
        yara_ruleset = compute_rules(goodwares, sub_algorithm, reports, features_list, ignore_fp)
        if yara_ruleset is None:
            """ Running again the UDT search with samples_size equal to 1, so to consider all the possible features."""
            samples_size = 1
            return createUDT(reports, sub_algorithm, goodwares, features_list, samples_size)
        else:
            restore(reports)
            return yara_ruleset
```

Figure 3.6: Clustering tree features selection.

where l is the feature that splits the partition A, $\mu(A_l)$ and $\mu(A_{\bar{l}})$ are the sum of the weights associated to each sample of, respectively, A_l and $A_{\bar{l}}$, and $g(A_l)$ and $g(A_{\bar{l}})$ are the centroids of the respective partitions.
When the samples have all the same weight, equal to 1, as for this thesis' context, $\mu(A_l)$ and $\mu(A_{\bar{l}})$ are equivalent to the total number of samples of the respective partition, $n(A_l)$ and $n(A_{\bar{l}})$. As a consequence, the previous expression becomes:

$$B(A_l, A_{\bar{l}}) = \frac{n(A_l)n(A_{\bar{l}})}{n(A_l) + n(A_{\bar{l}})} distance^2(g(A_l), g(A_{\bar{l}})).$$

The "Unsupervised Decision Tree"[6] algorithm, on the contrary, proposes 4 different measures, two of which are computationally expensive but provide accurate selection of the feature, while the others are less accurate but faster.
Both the approaches have justified measures selection, but, for the purposes of selecting a proper stopping criterion, they are not well suited. In particular, none of the measures there mentioned have an intuitive threshold that indicates whether the cluster that is being split is already sufficiently homogeneous.
For this reason, **the splitting criterion chosen for the current implementation**

relies on the maximization of a distance parameter.
Specifically, **the distance that is maximized at each step is the one between the closest binary approximation of the centroids of the clusters**. In particular, the approximation of the centroids is obtained by rounding the centroids' features values, which, being the mean of all the feature values of the cluster they represent, are typically not integer (Figure 3.7).

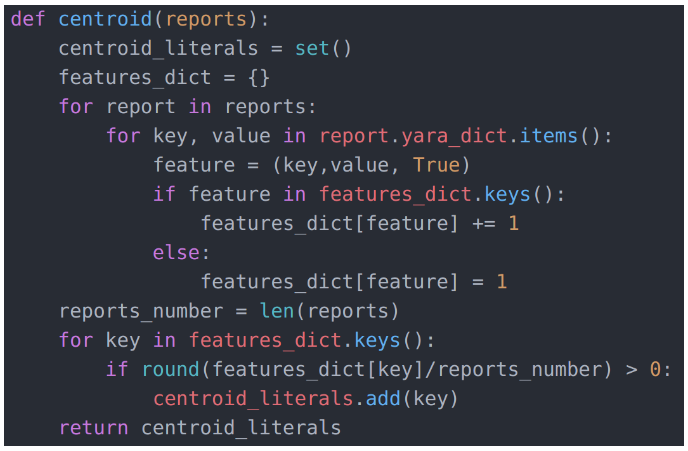

```python
def centroid(reports):
    centroid_literals = set()
    features_dict = {}
    for report in reports:
        for key, value in report.yara_dict.items():
            feature = (key,value, True)
            if feature in features_dict.keys():
                features_dict[feature] += 1
            else:
                features_dict[feature] = 1
    reports_number = len(reports)
    for key in features_dict.keys():
        if round(features_dict[key]/reports_number) > 0:
            centroid_literals.add(key)
    return centroid_literals
```

Figure 3.7: Function that determines the closest binary feature point to the centroid of a cluster.

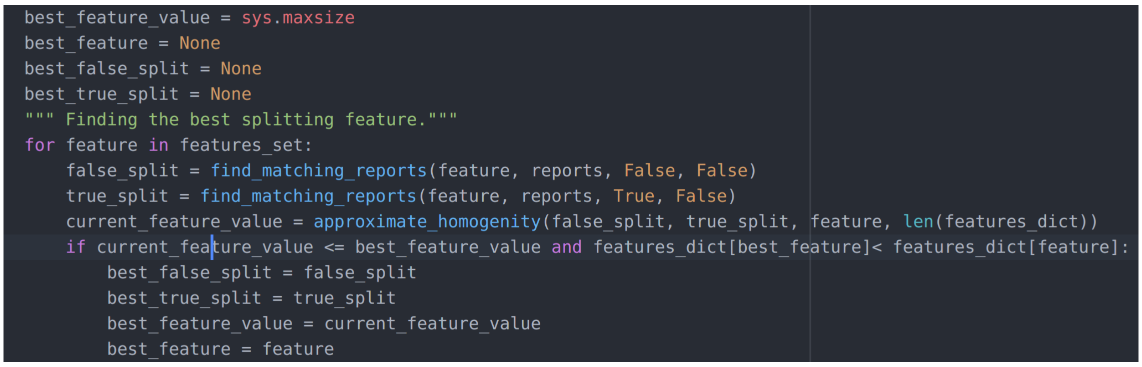
The rounding of the centroids has the benefit of making possible the application of the Jaccard and the Russell Rao measures while smoothing the weight of the highly variable integer and string features. Indeed, many features, such as the entrypoint one, or the section names ones, tend to assume several different values. When converting these features to binary ones, there will be one new binary feature per value. If one particular value of these is relevant for the cluster, this would be the prevalent one among all the possible values, and, when computing the cluster's centroid, it will likely have a corresponding mean value close to 1. On the other hand, a feature that assumes an approximately homogeneous distribution of the feature's values will have no prevalent value, so, all the binary features averages in the centroid will be close to 0.
The rounding of each feature allows then to enhance the weight of common string and integer feature values and to reduce the weight of rare or not relevant ones.
In mathematical terms, given a cluster's centroid $g(A)$, for which each feature's value is a real number between 0 and 1, and its features set F, the binary approximation of the centroid $a(A)$ is determined by:

$$\forall i \in F,\ a_i(A) = \begin{cases} 1 & \text{if } g_i(A) >= 0.5 \\ 0 & \text{otherwise} \end{cases}$$

The feature selected when splitting a cluster A is then:

$$l = argmax_{i \in F}\{distance(a(A_i), a(A_{\bar{i}}))\},$$

where A_i and $A_{\bar{i}}$ are the two partitions determined, respectively, by the samples that have the feature i (i.e., each sample s such that $s_i = 1$) and by the samples that don't have the feature i.
Currently, if there is more than one feature that has the maximum distance value, the selection of the splitting one is a random choice, however, it might be possible to use additional and more precise criteria, as it is done in [16].
The procedure that selects the best feature and its partitions is in Figure 3.8.



```
best_feature_value = sys.maxsize
best_feature = None
best_false_split = None
best_true_split = None
""" Finding the best splitting feature."""
for feature in features_set:
    false_split = find_matching_reports(feature, reports, False, False)
    true_split = find_matching_reports(feature, reports, True, False)
    current_feature_value = approximate_homogenity(false_split, true_split, feature, len(features_dict))
    if current_feature_value <= best_feature_value and features_dict[best_feature]< features_dict[feature]:
        best_false_split = false_split
        best_true_split = true_split
        best_feature_value = current_feature_value
        best_feature = feature
```

Figure 3.8: Portion of code that determines the best splitting feature for the provided dataset.

Evaluation of the stopping criterion and rules generation

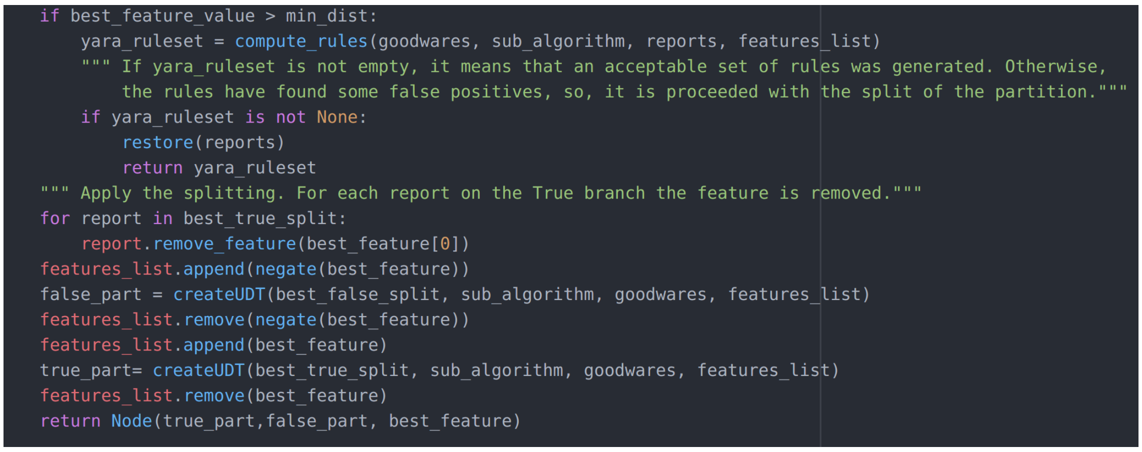
Once the best splitting feature has been defined, the two new sub-clusters are determined by taking the two partitions A_l and $A_{\bar{l}}$ associated to that feature.
After the sub-clusters have been found, it is necessary to evaluate whether they can be considered valid or if they are too close each other. To do this, it is fixed a distance threshold, that defines when the two clusters can be preserved and when the rules generation can start over the entire partition $A = A_l \cup A_{\bar{l}}$. In particular, the split is stopped when:

$$distance(a(A_l), a(A_{\bar{l}})) < min\_threshold.$$

This stopping criterion justifies the usage of the plain distance measure as splitting quality evaluation, since it gives an intuitive idea of which should the threshold be (approximately).
Experimental results have shown that, when using the Russell Rao similarity, a distance threshold of 0.7 is accurate enough to generate good clusters. Similarly, for the Jaccard similarity it has been set a distance threshold of 0.45.
The chosen values have an heuristic motivation behind:

- when the centroids' approximations have a distance of less than 0.7, using the Russell Rao similarity, it means that, among all the possible features, at least 30% of them are shared;
- when the centroids' approximations have a Jaccard distance of less than 0.45, it means that, among all the features possessed by at least one of the two representatives, at least 55 % of them are shared.

Both the just mentioned percentages are such that, when generating the rules, a sufficiently high number of features per rule is selected (around 100 features per rule in the worst cases).
The portion of code that evaluates the stopping criterion and acts consequently is in Figure 3.9.



```
if best_feature_value > min_dist:
    yara_ruleset = compute_rules(goodwares, sub_algorithm, reports, features_list)
    """ If yara_ruleset is not empty, it means that an acceptable set of rules was generated. Otherwise,
        the rules have found some false positives, so, it is proceeded with the split of the partition."""
    if yara_ruleset is not None:
        restore(reports)
        return yara_ruleset
""" Apply the splitting. For each report on the True branch the feature is removed."""
for report in best_true_split:
    report.remove_feature(best_feature[0])
features_list.append(negate(best_feature))
false_part = createUDT(best_false_split, sub_algorithm, goodwares, features_list)
features_list.remove(negate(best_feature))
features_list.append(best_feature)
true_part= createUDT(best_true_split, sub_algorithm, goodwares, features_list)
features_list.remove(best_feature)
return Node(true_part,false_part, best_feature)
```

Figure 3.9: Last part of the tree generation code. It includes the evaluation of the stopping criterion and the rules generation. As for Figure 3.8, the stopping disequation is here reverted due to the signs of the distance measures.

Finally, if the two sub-clusters are close enough (i.e., the stopping criterion is verified), the rules generation procedure starts. As it was happening for the HDBSCAN clustering, if the rules generated match any goodware the user provides, the returned set of rules will be empty. In this case, the sub-clusters determined at the previous steps will be considered valid and the procedure starts (recursively) again on each of them.

The logical tree structure. The bi-partition created by the algorithm, by splitting on a single feature, identifies a tree whose leaves are the final clusters. However, the main reason why the tree was chosen is the simultaneous generation of part of the YARA rules while clustering.
Specifically, as Figure 3.9 shows, just before recurring in any of the two sub-partitions, the feature that generated the split is added to a list of current features, eventually negated

when it is being considered the partition that does not have that feature. Moreover, that feature is removed from the cluster that is determined by that feature's presence, so that the subsequent steps of the procedure focus only on novel features.
The features list just introduced identifies the cluster's path in the clustering tree (from the root to the leaf) where, being the clustering monothetic, each branch is characterized by the presence or absence of the node's feature for that branch samples. In other words, the list of features at each leaf (i.e., cluster) represents a set of features whose presence or absence is shared for the points of the cluster. For this reason, these features, together with their boolean value, can be added to the YARA rules generated on that cluster.
A graphical representation of the tree produced when applying the "Unsupervised Decision Tree" algorithm on the "scar" family is in Figure 3.10.

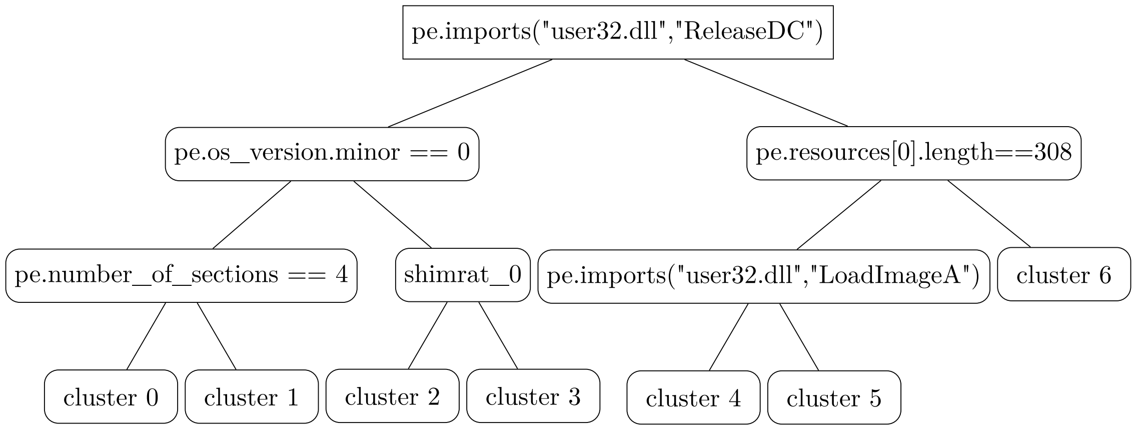
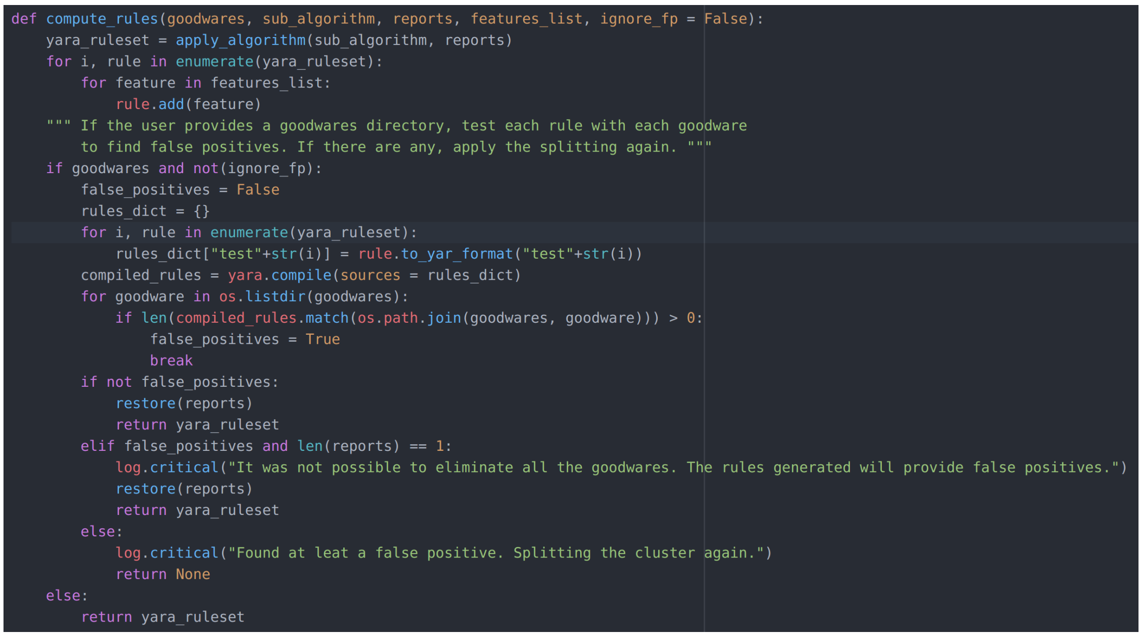

pe.imports("user32.dll","ReleaseDC")
pe.os_version.minor == 0
pe.resources[0].length==308
pe.number_of_sections == 4
shimrat_0
pe.imports("user32.dll","LoadImageA")
cluster 6
cluster 0
cluster 1
cluster 2
cluster 3
cluster 4
cluster 5

Figure 3.10: Resulting tree of the "Unsupervised Decision Tree" algorithm applied on the "scar" family. Each node indicates the splitting feature while the two branches indicate, respectively, on the left that the feature is True for the subset, on the right that the feature is False.

3.4 Rules generation

The rules generation is the last step of the tool. This step relies on a function already mentioned multiple times, that is "compute_rules" . This function has multiple tasks but relies on the underlying algorithm the user has chosen.
The actual version of the tool supports two different algorithms, the *greedy* and the *clot* algorithm, which will be described in this section.
Back to the "compute_rules" function, its body is almost equivalent in the HDBSCAN and the "Unsupervised Decision Tree" algorithms, so, in the following lines, it will be described just once, underlying the differences when needed. The current implementation of the function is depicted in Figure 3.11.
The function starts by calling the algorithm requested by the user on the samples contained



```python
def compute_rules(goodwares, sub_algorithm, reports, features_list, ignore_fp = False):
    yara_ruleset = apply_algorithm(sub_algorithm, reports)
    for i, rule in enumerate(yara_ruleset):
        for feature in features_list:
            rule.add(feature)
    """ If the user provides a goodwares directory, test each rule with each goodware
        to find false positives. If there are any, apply the splitting again. """
    if goodwares and not(ignore_fp):
        false_positives = False
        rules_dict = {}
        for i, rule in enumerate(yara_ruleset):
            rules_dict["test"+str(i)] = rule.to_yar_format("test"+str(i))
        compiled_rules = yara.compile(sources = rules_dict)
        for goodware in os.listdir(goodwares):
            if len(compiled_rules.match(os.path.join(goodwares, goodware))) > 0:
                false_positives = True
                break
        if not false_positives:
            restore(reports)
            return yara_ruleset
        elif false_positives and len(reports) == 1:
            log.critical("It was not possible to eliminate all the goodwares. The rules generated will provide false positives.")
            restore(reports)
            return yara_ruleset
        else:
            log.critical("Found at leat a false positive. Splitting the cluster again.")
            return None
    else:
        return yara_ruleset
```

Figure 3.11: Function that generates the YARA rules. It relies on the algorithm asked by the user and, if the user provided any goodware, it checks that the rules obtained do not generate false positives. If they do, then it returns a void rule set.

in the considered cluster. Then, *in the case of the "Unsupervised Decision Tree"*, the set of features that characterize the cluster (i.e., the ones that are determined at the clustering step) are added to each produced rule.

Finally, if the user provided some goodwares, each of them is tested against the rules and, if at least a false positive is found and the cluster that generated the rules has at least two samples, it is returned a void rule set (None value). In any other case, the set of rules determined by the previous steps is returned to the caller, either because the rules did not generate any false positive, or because the cluster was consisting of a single element, which means, no more clustering is possible.

3.4.1 The underlying algorithms

The greedy and the clot algorithms were initially developed for the YaYaGen tool, however, they work on any general set of features, so, they were both suitable for the Portable Executable tool variant.

In the following sections, there will be a description of the general idea of both the procedures, not referring to the particular YARA generation procedure, but considering the goal of finding a group of features sets that can cover all the data samples.

In mathematical terms: lets consider a set of data samples S. Each sample $s \in S$ is characterized by a subset F_s of features, where $F_s \subset F$ and F is the overall set of features.

Alternatively, each sample $s \in S$ could be seen as an array of $|F|$ elements such that:

$$\forall\ f \in F,\ s_f = \begin{cases} 1 & \text{if s has feature f} \\ 0 & \text{otherwise} \end{cases}$$

Each feature $f \in F$ is associated with its relative relevance score w which indicates the importance of the feature's presence in a data sample.
From the set of features F, it is possible to identify its powerset $\mathcal{P}(F)$. The idea of both the algorithms is to try to find, from the powerset of F a "good" set of elements (i.e., sub-sets of F) $R \in \mathcal{P}(F)$ such that:

$$\forall\ s \in S\ \exists\ r \in R:\ \forall\ f \in r,\ s_f = 1.$$

In other words, the algorithms try to find a "good" set of rules R that covers completely the samples set S, in which, each rule $r \in R$ is composed of a set of features $f \in F$.
Alternatively, these algorithms can be seen as a sub-optimal solution of the "Set cover" problem [53] where the set of rules R is the solution that covers the entire universe, which corresponds to the entire set of data samples.

The greedy algorithm

The greedy algorithm tries to do what just mentioned by searching for the first local optimum solution it can get.
The algorithm requires a threshold t that guides the search of "good" rules, and that can be manually configured by the user, so to find the best balance between rules specificity and number of generated rules.
The algorithm iterates on all the samples not covered by any rule. In particular, it determines two elements in the set of uncovered samples such that the common features' score is maximum, by relying on the score of each feature previously mentioned (procedure in Algorithm 3).
Once the best intersection of samples is found, the corresponding set of features F_i is tested against all the remaining uncovered samples, and, if any matching sample is found, it is removed from the list of the currently uncovered samples.
For all the remaining samples, it is then progressively identified a new set of features F_r by finding the best intersection between the current set of features F_i and those of any other sample (Algorithm 4).
The procedure goes on until it is found a set of features that has a score lower than the fixed threshold t. When this happens, the set of uncovered samples is updated, by removing the samples s such that $F_s \supseteq F_r$, and the procedure starts again, until the set of uncovered samples is empty.
The complete procedure is in Algorithm 5.

Algorithm 3 Function that determines the best couple of samples to get features from. The evaluate function is used to measure the quality of the common features of the samples.

```
function EVALUATE($F, W$)
    $val = 0.0$
    for $f \in F$ do
        $val = val + W[f]$
    end for
    return val
end function

function GET_BEST_FEATURE_PAIRS($U$)
    $best\_rule \leftarrow \emptyset$
    $best\_val \leftarrow 0.0$
    for $s_i \in U$ do
        for $s_j \in U \setminus \{s_i\}$ do
            $features = s_i \cap s_j$
            val = EVALUATE(features, W)
            if $val > best\_val$ then
                $best\_val = val$
                $best\_rule = features$
            end if
        end for
    end for
    return $best\_rule$
end function
```

Algorithm 4 Function that finds the best intersection between a predefined set of features and any other sample in a set of samples L.

```
function GET_RELAXED_RULE(yara_rule, L, W)
    $best\_rule \leftarrow \emptyset$
    $best\_val \leftarrow 0.0$
    for $s \in L$ do
        $features = yara\_rule \cap$ FEATURES($s$)
        val = EVALUATE(features, W)
        if $val > best\_val$ then
            $best\_val = val$
            $best\_rule = features$
        end if
    end for
    return $best\_rule$
end function
```

Algorithm 5 Greedy algorithm pseudo-code. It takes as parameters a samples set S and the user defined threshold t

```
function GREEDY($S, W, t$)
    $U \leftarrow S$
    $rules\_list = List()$
    while $U \neq \emptyset$ do
        if $|U| > 1$ then
            yara_rule = GET__BEST__FEATURE__PAIRS($U$)
        else
            yara_rule = GET__FEATURES(U)
        end if
        $L \leftarrow U \setminus \{s \in U : yara\_rule \subseteq \text{FEATURES}(s)\}$
        while $L \neq \emptyset$ do
            new_rule = GET__RELAXED__RULE(yara_rule, L, W)
            if $new\_rule = \emptyset$ then
                new_rule = yara_rule
            end if
            if EVALUATE($new\_rule, W$) $< t$ then
                break
            end if
            yara_rule = new_rule
            $L \leftarrow U \setminus \{s \in U : yara\_rule \subseteq \text{FEATURES}(s)\}$
        end while
        $U \leftarrow L$
        $rules\_list += yara\_rule$
    end while
end function
```

The clot algorithm

The clot algorithm is more complex than the greedy counterpart, however, it does not search for the first greedy solution possible but tries to generate a smarter (and qualitatively better) set of rules, at the cost of higher computational time.

In this section, for readability reasons, it is referred to any features set as a rule, implying the subsequent conversion process of each set to a YARA rule.
The *clot* algorithm, as opposed to the greedy procedure, starts by creating a first set of valuable rules from which picking the best ones afterwards.
The set of rules just mentioned is extracted by initially generating one rule per sample, as specific as possible. Then, for each rule in the set, it is determined a new rule as the best intersection of this rule's features with any other rule. If the resulting rule has a score higher than the user specified threshold t, it is then added to the set of rules (Algorithm 6).

Algorithm 6 Function that determines the initial set of rules for the set of samples S from which extracting the best ones. The "coverage" function referred is in charge of evaluating how many samples, in the set S, are covered by the rule given as parameter.

```
function FIND_RULESET($S, W, t$)
    $heap = List()$
    $bunch = List()$
    $rules\_list = List()$
    for $s \in S$ do
        $rule =$ FEATURES($s$)
        $heap$.APPEND($rule$)
        $bunch$.APPEND($rule$)
    end for
    while $|heap| > 0$ do
        $heap =$ SORT($heap$)
        $rule1 = heap$.EXTRACT_LAST_ELEMENT()
        $best \leftarrow \emptyset$
        for $rule2$ $in$ $bunch$ do
            $new\_rule = rule1 \cap rule2$
            if new_rule in bunch then
                continue
            end if
            if EVALUATE($new\_rule, W$) $\geq t \wedge best == \emptyset$ then
                $best = new\_rule$
            end if
            if EVALUATE($new\_rule, W$) $\geq t \wedge$
              (COVERAGE($best, S$)<COVERAGE($new\_rule, S$) $\vee$
              COVERAGE($best, S$)==COVERAGE($new\_rule, S$) $\wedge$
              EVALUATE($best, W$)>EVALUATE($new\_rule, W$))     then
                $best = new\_rule$
            end if
        end for
        if $best \neq \emptyset$ then
            $heap$.APPEND($best$)
            $bunch$.APPEND($best$)
        end if
    end while
    return bunch
end function
```

This procedure goes on until all the rules in the set, including those generated meanwhile, have been taken into account.
With this methodology, the algorithm is capable of looking up for all the best intersections possible, eventually considering multiple intersections, but the decision on which of these

are part of the final ruleset is taken afterwards, that is, considering all these combinations. In this sense, the algorithm is not strictly greedy, but has some degree of movement with respect to the previous one.
The choice of the final set of rules is done in two separate steps: a preliminary one determines a greedy set of rules while the second one, available only for relatively small rule sets, searches for the optimal combination of rules.
The preliminary step iterates until all the samples are covered by at least a rule. In particular, at each iteration, it selects the sample that is covered by the smallest number of rules. Among the rules that cover this sample, the procedure selects the one with the highest coverage of the uncovered samples and adds it to the final rule set. Once the rule has been added to the rule set, all the samples that are covered by that are removed from the set of uncovered samples, and the iteration starts again (Algorithm 7).

Algorithm 7 Function that determines a reasonable solution for the set of samples S, from the rule set R provided at the previous steps. The “FIND_BEST_RULE” function searches for the best rule in the set provided as parameter, privileging the highest coverage value and, in case of equality, the smallest rule score.

```
function FIND_REASONABLE_SOLUTION($S, R, W$)
    $U \leftarrow S$
    $ruleset = List()$
    while $|U| > 0$ do
        $critical\_set \leftarrow \emptyset$
        for s in U do
            $current\_set =$ FIND_MATCHING_RULES($s, R$)
            if $critical\_set == \emptyset \ \vee \ |current\_set| < |critical\_set|$ then
                $critical\_set = current\_set$
            end if
        end for
        $rule =$ FIND_BEST_RULE($critical\_set$)
        $ruleset$.APPEND($rule$)
        $U \leftarrow U \setminus \{s \in U : rule \subseteq \text{FEATURES}(s)\}$
    end while
    return ruleset
end function
```

At the end of the procedure, there will be a final set of rules that is capable of covering all the samples, however, this does not necessarily provide the best solution possible, since it selects rules by focusing first on the least covered samples.
For this reason, when the set of rules is fairly small, it is reasonable to try to find the best combination of rules possible. This search is done by considering all the combinations of the rules, to find the best combination that covers all the samples.
Ideally, this procedure should take into account the analysis of the power set of the set

of rules, however, this is usually computationally too expensive. For this reason, the algorithm starts searching for the best solution on all the combinations of length n, where n is the size of the rule set found at the preliminary step.
If any optimal solution is found on these combinations, it is then searched for the best combination of *n-1* rules. This procedure goes on, progressively reducing the number of rules in each combination, until it cannot find an optimal solution anymore(Algorithm 8).

The complete sequence of function calls that drive the overall clot algorithm is in Algorithm 9, which contains also the directive that decides when to apply the research for an optimal solution and when to accept directly the first one found.

Converting the features sets to YARA rules.

Both the algorithms presented return, as a final solution, multiple features sets. These sets, however, have to be accurately translated to a YARA rule each.
This translation mechanism is performed by simply creating, for each features set, a YARA rule whose condition contains the *conjunction* (i.e., the “AND”) of all the features in the set. Eventually, if the user provided some YARA rules to be included as features, a *private YARA rule* for each selected rule feature is created.
In the end, the *disjunction* (i.e., “OR”) of all the (*not private*) YARA rules thus created constitutes the rule-set covering the input samples.

Algorithm 8 Function that determines the optimal solution for the set of samples S, from the rule set R provided at the previous steps, privileging the highest coverage value and, in case of equality, the smallest ruleset score. “Coverage” and “Evaluate” functions are here the sum of the “Coverage” and “Evaluate” functions applied for each rule in the ruleset.

```
function FIND_BEST_SOLUTION($S, R, W, target\_size$)
    $yara\_ruleset \leftarrow \emptyset$
    while $run\_exact == TRUE$ do
        $best\_solution \leftarrow \emptyset$
        if target_size == 0 then
            break
        end if
        $rules\_combination$ = GET_COMBINATIONS($R, target\_size$)
        for combination in rules_combination do
            $U \leftarrow S$
            for rule in combination do
                $U \leftarrow U \setminus \{s \in U : rule \subseteq$ FEATURES$(s)\}$
            end for
            if $|U| > 0$ then
                continue
            end if
            if $best\_solution == \emptyset \vee$
                (COVERAGE($best\_solution, S$)<COVERAGE($combination, S$) $\vee$
                COVERAGE($best, S$)==COVERAGE($combination, S$) $\wedge$
                EVALUATE($best\_solution, W$)>EVALUATE($combination, W$))  then
                $best\_solution = combination$
            end if
        end for
        if $best\_solution \neq \emptyset$ then
            yara_ruleset = best_solution
            $target\_size = |yara\_ruleset| - 1$
        else
            $run\_exact = FALSE$
        end if
    end while
    return yara_ruleset
end function
```

Algorithm 9 Clot algorithm pseudo-code. The function has been expressed as the sequence of calls to the previous functions, for readability reasons.

```
function CLOT($S, W, t$)
    $yara\_ruleset \leftarrow \emptyset$
    $initial\_ruleset$ = FIND_RULESET($S, W, t$)
    $yara\_ruleset$ = FIND_REASONABLE_SOLUTION($S, initial\_ruleset, W$)
    $target\_size = |yara\_ruleset|$
    $search\_space\_size$ = GET_NUMBER_OF_COMBINATIONS($|initial\_ruleset|, target\_size$)
    if $search\_space\_size > 500$ then
        return yara_ruleset
    else
        $optimal\_ruleset$ = FIND_BEST_SOLUTION($S, initial\_ruleset, W, target\_size$)
        if $optimal\_ruleset \neq \emptyset$ then
            $yara\_ruleset = optimal\_ruleset$
        end if
    end if
    return yara_ruleset
end function
```

Chapter 4

Experimental results.

This chapter presents the experimental results obtained with the tool, using the different distance measures and algorithms and trying different parameters combinations and data samples.
The dissertation covers different aspects of the experimental results, and uses, as comparison meters, the yaBin and yarGen tools, introduced in subsection 2.3.2. The remaining YARA rule generation tools are not reported here since they have been showing poor results in most of the tests faced.
Although it has been tried to give a reasonable and sound set of results, in the context of malware identification, outcomes are tricky to evaluate. In particular, it is not a trivial task to determine the family of a given sample, and, as a consequence, working on families of malwares does not necessarily provide results that have high precision, because they might depend a lot on the mis-classification of some samples' families.
The mis-classification of a sample's family, indeed, might reduce the quality of the results, because there might be samples and, consequently, rules that are actually belonging to other families. For this reason, in most of the chapter, it will be taken into account a relative comparison among the tools, rather than evaluating absolutely the tool's quality.
To this extent, the experiments focused on getting an idea of 5 different aspects, covered from section 4.2 to section 4.6:

- the quality of the clustering procedures and the capability of clustering using the current features set;
- the capability of recognizing malware families and not their packers;
- the capacity of the tool of avoiding false positives;
- the capability of the rules to find malicious samples they weren't trained on, even of different families;
- the capability of the tools to cover the entire training set and the number of samples covered per rule.

Finally, section 4.7 presents a brief discussion of the previous aspects considered all together.
The last two sections focus, instead, on an in-depth analysis of the YaYaGenPE performances. Specifically, section 4.8 presents the results obtained by testing some of the generated rules over the VirusTotal Intelligence platform, while section 4.9 reports the results of the k-fold validation performed on some of the biggest malware families in the dataset.

4.1 Dataset Analysis

This section provides an insight on the dataset provided by VirusTotal. The initial study is based on a dataset consisting of 6,881 malicious executables. Each sample is associated with the VirusTotal report, which indicates all the most relevant information and the outcome of the analysis performed by about 30 relevant Anti-Virus products.

4.1.1 Families detection

In order to extract the most probable family for each sample in the dataset, it has been used AVclass, a pre-trained classifier. This tool, relying on an internal training set, associates the most likely family to each given sample. The detailed approach of the tool is reported in [52]; briefly, the tool uses the labels provided by Anti-Viruses on each report in order to find out the most common family. In order to do this, the process performs a procedure of cleaning and standardization of each Anti-Virus label, for which it relies on two internal lists, one to clean the labels and produce a set of tokens, the other to remove family aliases. This is needed because Anti-Viruses do not report directly the family name, but, each match, consists of the family plus other spurious information. Moreover, different Anti-Viruses tend to give different names to the same family, so, before ranking the most frequent families, aliases of each have to be converted to a single name.
This classification process has some limitations and inaccuracies, in particular in the detection of family aliases and when it has to choose the most likely family among a set of ex-aequo ones, for which the choice is almost randomic. Nevertheless, given the size of the dataset, it is not feasible to proceed on a manual analysis in a reasonable amount of time, thus, automatic classification is necessary.
Figure 4.1 illustrates the distribution of families in the dataset.

4.1.2 Family samples similarities

For a further analysis, each family was grouped and, for each of them, ssdeep and imphash have been used to evaluate the similarity of every sample with any other of the same family. The former is based on fuzzy hashing in order to determine similarity of two files [32]. The latter is the same function introduced in section 3.2.1 and, as an hashing function, it is

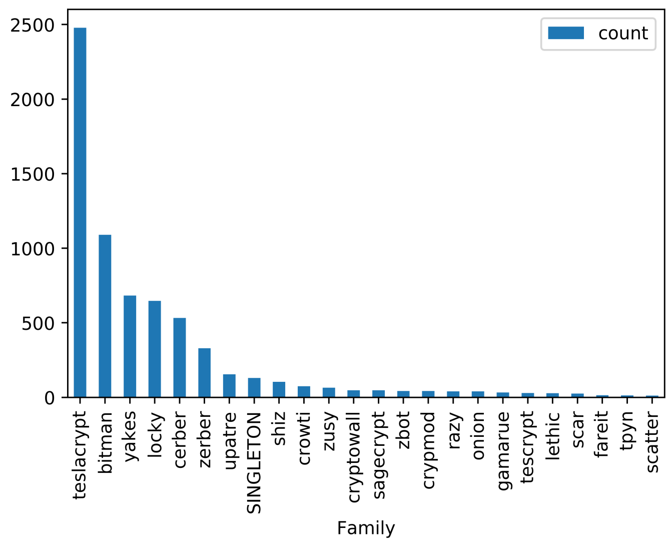


Figure 4.1: Distribution of families with more than 10 samples for the EXE dataset.

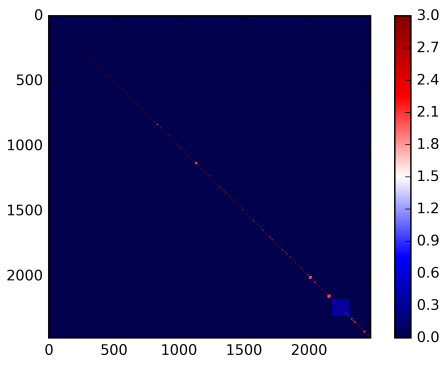
only an indicator of perfect match.
None of the two metrics is accurate, since ssdeep can find only files really close each other and imphash is an heuristic for which even a simple swapping of DLLs would imply not matching anymore. Nevertheless, for lots of families, the complementary usage of those two metrics allows to detect similarities in the analysed samples. Figure 4.2 illustrates the samples similarities on simmetric heatmaps of some of the most relevant families.

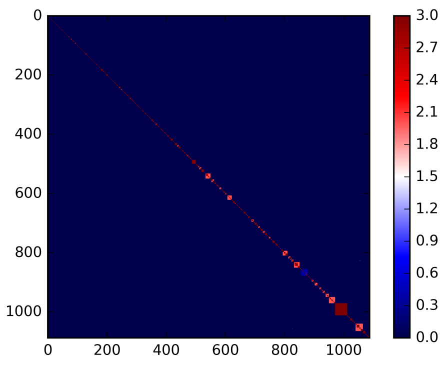
4.1.3 The YARA rule-set

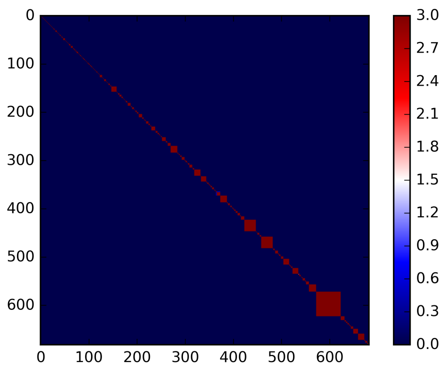
To get a better picture of the dataset under-analysis, it has been made use of a big ensemble of rules, coming from several open-source github repositories [4]. After a cleaning phase, in which rules that did not aim at PE files or not matching any sample or syntactically wrong were removed, the remaining rules have finally been grouped into two categories: *malware matching* rules, that search for malicious behavior, and *packers matching* rules, that look for packed samples.
The results of the application of these rules to the mentioned dataset are the followings:

- **Malware rules**: 68.8%;
- **Packers rules**: 63.4%.

Rules matching percentages for malware is in the order of 70%, even though, looking at the VirusTotal reports, the number of Anti-Virus positives is high (more than 50% of Anti-Viruses detect samples as malicious in almost all samples). This might be related to the high percentage of packed samples: being most of them packed and being the rules based


(a) Teslacrypt family.


(b) Bitman family.


(c) Yakes family.

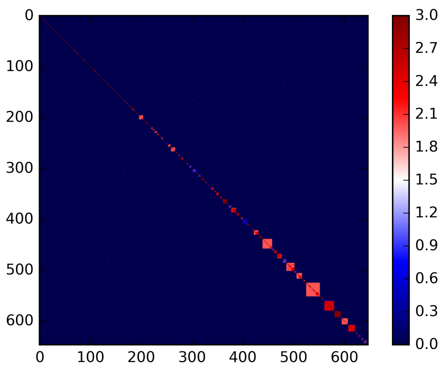

(d) Locky family.

Figure 4.2: Heatmaps that show the correlation between samples of some EXE families. Disposition of samples has been purposely rearranged to collapse together similar samples as much as possible. Following the legend: colors from 0.0 to 1.0 indicate the degree of similarity of samples for what concerns ssdeep, while imphash is different; color at 2.0 indicates samples equivalent for imphash measure; colors from 2.0 to 3.0 indicate that samples are equivalent from the imphash point of view (value 2.0) and that they have a degree of similarity for what concerns ssdeep as well (ssdeep similarity > 0.0).

on static analysis, it is highly probable that most of rules are not capable of matching the samples. Moreover, it is important to remark that most of the packers matching rules are detecting commercial and widely spread packers, so, it is likely that the number of packed samples is even higher than the percentages indicated previously.
From Table 4.1 it is possible to see how a big percentage (approximately 37%) of samples is packed using Armadillo, which is one of the strongest packers commercially distributed. This should drive the analysis towards characteristics of the executables that are not so much affected by packing.

Packer	Matched samples
Armadillo	2537
NSIS	338
PureBasic	105
PCGuard	95
aPLib	58
PECompact	48
Yoda	30
UPX	27
ASProtect	12
ASPack	5
VMProtect	4

Table 4.1: Table showing the number of samples in the dataset of executables (EXE) for each matching packer. It is worth mentioning NSIS, which is not, usually, a packer, but an installer, however, as described in [14], some ransomwares have been extensively misusing it as a packer.

4.2 Clustering evaluation

A question that is reasonable to ask, when talking about clustering of malicious samples, is whether the clusters are coherent enough to represent (generations of) families or not. This question is even more suitable when dealing with static analysis, for which the behavior of a family is not taken into account (at least directly).
Most of the results in this work have been obtained by considering separated families, already divided by using the labels given by the AVclass tool. However, this tool is limited to determining the overall family and it is not capable of giving details about the family's generation and variant. For this reason, it was not possible to test the accuracy of clustering when operating on a single family, so, the tests focused on considering all the families together and on evaluating the clusters with respect to the family labels inside each cluster.
In order to evaluate clustering quality, it has been used the V-measure [45]. This particular measure, similarly to the F-measure applied for classification algorithms, consists of the harmonic mean of two other measures, that are *homogeneity* and *completeness*. As Rosenberg and Hirschberg explained [45], typically, the two measures are opposed one to each other, i.e., when homogeneity increases, completeness tends to decrease and vice-versa.
In particular, the *homogeneity* measure is an indicator of how much are the clusters homogeneous, i.e., how much does the algorithm tend to group together data samples having the same data labels.
The *completeness* measure, on the other hand, measures how much the algorithm is capable of clustering all the data samples having a given label inside the same cluster.

These measures, to work properly, need an already labeled solution. For this work purposes, the labeled solution used was the AVclass labeling output, which associates, to each sample, its most likely family.
However, this solution, as based on the most common family attributed by all the Anti-Viruses that analyzed the sample, is not precise. Specifically, AVclass does not take into account the accuracy and relevance of the Anti-Virus that attributed a label and, even more important, it is not capable of dealing properly with ex-aequo of family labels, ultimately ending in a random choice of the label.
Moreover, although AVclass developers have made a good job in aliases removal, it might still happen that some families are aliases of some others (e.g., "teslacrypt" is also named "bitman" by kaspersky labs [31], and "cerber" family is associated with the "zerber" family as well [30]). Furthermore, some families might be usually associated with others since one is a downloader of the other(s) (e.g., the "upatre" family is part of the downloaders class and it typically drops families like "cryptowall", "crowti" and "zbot"). These limitations might impact negatively the clustering results, since the label chosen by AVclass might be the wrong one, or a non-detected alias of others in the same cluster.
Homogeneity and completeness of all the data samples are in Table 4.2.

Algorithm	Homogeneity	Completeness
UDT	0.82	0.28
UDT+rules	0.81	0.28
HDBSCAN	0.79	0.28
HDBSCAN+rules	0.80	0.28

Algorithm	Homogeneity	Completeness
UDT	0.72	0.29
UDT+rules	0.74	0.29
HDBSCAN	0.89	0.27
HDBSCAN+rules	0.92	0.28

Table 4.2: Tables representing the Homogeneity and Completeness of the entire dataset, computed when clustering using all the possible combinations of features and clustering algorithms. Table on the left presents results when using the Russell-Rao distance, while, on the right, results were obtained using the Jaccard distance.

The results evidence a fairly high homogeneity value while a rather small completeness. This outcome is exactly what was expected from the measures definitions, however, it is not much concerning the completeness low values, since it is not usually the case that each family has to be in a single cluster. As already mentioned in chapter 2, metamorphic and polimorphic engines generate a huge variety even in samples belonging to the same family. For this reason, especially when considering static analysis, it is no wonder that the same family is spread among several clusters.
The main focus of this analysis was, in fact, on the homogeneity value, which gives an idea of how well similar samples of the same family are grouped together.
Although results are not optimal, they show an acceptable clustering quality, especially considering the imprecise "ground truth" used as comparison meter (i.e., the AVclass labels). The goodness of the results was also manually verified, by checking that the majority of the clusters, in all the clustering procedures, contained up to two different families each.

Results show an opposite behavior of the “Unsupervised Decision Tree” and “HDBSCAN” algorithms. In particular, the former tends to generate more accurate results using the Russell-Rao measure, while the latter produces better (and good) results with the Jaccard distance.
As a matter of fact, these results might be somehow misleading. In particular, another information that has to be taken into account for the purposes of rules generation is the number of clusters created and the number of single point clusters among them.
Although this information is somehow contained in the *completeness* measure and does not account heavily in clusters evaluation, it is practically important to consider the number of clusters for this work, since it directly affects the number and the specificity of the generated rules. In particular, the higher the number of clusters, the higher will the rules specificity be. Ultimately, when dealing with single point clusters, the rule will be as specific as possible for a given sample, and this might lead to over specific rules. Table 4.3 represents the number of clusters found by each of the algorithms in Table 4.2 along with the number of clusters containing a single point and the average cluster size.

Algorithm	**# Clusters**	**Single Pts. clusters**	**Avg. cluster size**
UDT	1390	546	4,9
UDT + rules	1348	488	5,0
HDBSCAN	1232	259	5,5
HDBSCAN + rules	1314	287	5,1

Algorithm	**# Clusters**	**Single Pts. clusters**	**Avg. cluster size**
UDT	1093	520	6,2
UDT + rules	1047	469	6,4
HDBSCAN	2124	524	3,2
HDBSCAN + rules	2245	572	3,0

Table 4.3: Tables showing the sizes of the clusters from which Table 4.2 was computed. Table on the left presents results when using the Russell-Rao distance, while, on the right, results were obtained using the Jaccard distance.

These results show the correlation between the number of clusters (and their size) and the *homogeneity* value: the higher is the former, the higher will the latter be. As said, although HDBSCAN with the Jaccard distance seems to produce the highest accurate clustering results, this clustering quality has to be weighted considering also the high number of produced clusters.
Finally, it is to be noted that, even though the other results show an homogeneity value that is from 0.72 up to 0.82 out of 1.0, the clustering procedure is just a preliminary step to make sure that the *greedy* or the *clot* algorithm can work on fairly similar samples, that

have not (but it would be preferable to) be perfectly clustered.

4.3 Packer vs. Family detection

One of the most important questions for this work was whether the rules generated detect the packers of the samples or their families. This question hasn't an easy and straightforward answer, since there are lots of different packers and they have different levels of complexity. Moreover, some of the packers have been modified by malware authors and their version is not publicly available.
Apart from the difficulties of finding the right packers to test,once a packer has been found, it is also difficult to determine the effective resilience of the rules to that packer.
A possible test to answer this question was performed by using the simplest packer openly diffused, i.e., UPX [63], and packing all the goodwares with it. The aim of the test is to check if the rules generated do match any of the packed goodwares.
For this test, only the families that had at least a sample packed with UPX, according to Table 4.1 were selected. In particular, excluding malwares that were not attributed to any family, the extracted families were: “cerber”, “locky”, “upatre” and “zerber”.
Of these families, the rules have been generated considering *all* the samples, even those that were not packed by UPX. The reason behind this choice is due to the fact that creating the rules only on UPX packed samples might bias the results.
These tests require a set of UPX packed goodwares. Finding a good set of UPX packed goodware is not easy, so, to get an approximately consistent set, it was applied the UPX packing procedure over the original database of 3,413 goodwares and it was selected the 3,028 goodwares that were successfully packed by UPX.

4.3.1 YaYaGenPE results

Table 4.4 shows the number of false positives for all the possible algorithms and training choices of the tool on the previously mentioned families when using the Russell-Rao distance. As shown, results are promising, in that, only one family (i.e., “Locky”) generated false positives on a single configuration. Further analysis also indicated that the 7 false positives found were due to a single rule, which was too generic, and presented a small number of condition literals, probably due to a not optimal cluster from which rule was generated.
It is worth mentioning that, when it is referred to “goodware” in Table 4.4, it is implied that the training procedure was done considering goodwares that were *not* packed by UPX. As a consequence, those rules did not generate any false positive on the “goodwares” set used during the training phase, but that does not prevent rules from finding false positives when the same set of goodwares is consequently packed. The rule that generated the false positives is shown in Listing 4.1.

Algorithms + parameters	Cerber	Locky	Upatre	Zerber
udt + greedy	0	0	0	0
udt + greedy + rules	0	0	0	0
udt + greedy + goodware	0	0	0	0
udt + greedy + rules + goodware	0	0	0	0
udt + clot	0	0	0	0
udt + clot + rules	0	0	0	0
udt + clot + goodware	0	0	0	0
udt + clot + rules + goodware	0	0	0	0
hdbscan + greedy	0	0	0	0
hdbscan + greedy + rules	0	0	0	0
hdbscan + greedy + goodware	0	0	0	0
hdbscan + greedy + rules + goodware	0	7	0	0
hdbscan + clot	0	0	0	0
hdbscan + clot + rules	0	0	0	0
hdbscan + clot + goodware	0	0	0	0
hdbscan + clot + rules + goodware	0	0	0	0

Table 4.4: Table reporting the number of false positives found on packed goodwares. Each row reports a combination of algorithms and parameters used for the training, while each column indicates the family in which the training was performed. Each entry indicates the number of false positives found by all the rules of the algorithm of the corresponding row when trained on the family of the corresponding column. All the rules obtained have been generated by using the Russell Rao distance.

```
import "pe"
import "math"
private rule APT17_Unsigned_Symantec_Binary_EFA_0{
        condition:
                pe.number_of_signatures == 0
}

private rule apt_ProjectSauron_encrypted_container_0{
        condition:
                math.entropy(0x400,filesize) >= 6.5
}


rule YaYaRule: rule97 {
        meta:
                author = "YaYaGen␣--␣Yet␣Another␣Yara␣Rule␣Generator␣for␣
                    PE␣files␣(*)␣v0.1_spring18"
                date = "14␣Aug␣2018"
                note = "Beta␣version"
        condition:
                pe.characteristics & pe.RELOCS_STRIPPED and
                pe.linker_version.minor == 0 and
                pe.checksum == 0 and
                pe.resource_version.major == 0 and
                pe.sections[2].characteristics & pe.SECTION_MEM_READ and
                pe.sections[0].characteristics & pe.SECTION_MEM_READ and
                pe.sections[1].characteristics &
                    pe.SECTION_CNT_INITIALIZED_DATA and
                pe.sections[0].virtual_address == 4096 and
                APT17_Unsigned_Symantec_Binary_EFA_0 and
                pe.sections[1].characteristics & pe.SECTION_MEM_READ and
                pe.checksum!=pe.calculate_checksum() and
                apt_ProjectSauron_encrypted_container_0 and
                pe.characteristics & pe.MACHINE_32BIT and
                pe.resource_timestamp == 0 and
                pe.sections[0].characteristics & pe.SECTION_MEM_EXECUTE and
                pe.image_version.major == 0 and
                pe.image_base == 4194304 and
                pe.imports("shell32.dll") and
                pe.subsystem == 2 and
                pe.sections[2].characteristics & pe.SECTION_MEM_WRITE and
                pe.imports("kernel32.dll") and
                pe.characteristics & pe.EXECUTABLE_IMAGE and
                pe.sections[2].characteristics &
                    pe.SECTION_CNT_INITIALIZED_DATA and
                pe.machine == 332 and
                pe.image_version.minor == 0 and
                pe.resource_version.minor == 0 and
                pe.imports("kernel32.dll","GetProcAddress")
}
```

Listing 4.1: Weak rule generated by the YaYaGenPE tool, which was causing false positives in packed goodwares.

As shown in Listing 4.1, the rule presents a small number of literals in the condition (a reasonable expected amount in these case has been experimentally deduced around 100 features or more) and the present literals are, for the most part, common features of both benign and malicious samples (e.g., any feature indicating read/write/execute characteristics of the PE sections).
Table 4.5 shows the results of the exact same test as the one just mentioned but over the rules produced using the “Jaccard” distance instead of the “Russell Rao” one.
As opposed to Table 4.4, in this case, no rule generated false positives. This can also be partially justified by the cluster statistics in Table 4.3, for which, considering “HDBSCAN” clustering, when using the “Jaccard” distance, more clusters were produced and, likely, more specific rules should result.
Comparing the number of generated rules in the two cases, it results that the former

Algorithms + parameters	**Cerber**	**Locky**	**Upatre**	**Zerber**
udt + greedy	0	0	0	0
udt + greedy + rules	0	0	0	0
udt + greedy + goodware	0	0	0	0
udt + greedy + rules + goodware	0	0	0	0
udt + clot	0	0	0	0
udt + clot + rules	0	0	0	0
udt + clot + goodware	0	0	0	0
udt + clot + rules + goodware	0	0	0	0
hdbscan + greedy	0	0	0	0
hdbscan + greedy + rules	0	0	0	0
hdbscan + greedy + goodware	0	0	0	0
hdbscan + greedy + rules + goodware	0	0	0	0
hdbscan + clot	0	0	0	0
hdbscan + clot + rules	0	0	0	0
hdbscan + clot + goodware	0	0	0	0
hdbscan + clot + rules + goodware	0	0	0	0

Table 4.5: Table reporting the number of false positives found on packed goodwares. The table follows the exact same structure of Table 4.4 and it has been obtained by using rules generated considering the “Jaccard” distance.

produced 210 rules, while the latter generated 237 rules.
Further analysis of the rules that detected false positives in Table 4.4 and their respective in Table 4.5 showed that the “rule97” previously introduced was one of the most generic, having only 27 features in its condition, while any rule produced using the “Jaccard” distance in the same situation had at least 58 features.
This big discrepancy in the number of features per condition is probably the main reason behind the difference in the false positives numbers, as adding features in a rule is likely to reduce the number of false positives that will be matched.

4.3.2 yarGen results

Even if these results are enough explicative of the rules efficacy, at least for what concerns UPX, Table 4.6 reports, as a comparison meter, the results obtained with another automatic YARA rule generator: yarGen. As shown in Table 4.6, except for the “upatre” family samples, any other family is reported to have false positives, on the majority of the tests performed.
As for YaYaGenPE, the “goodware” parameter used in yarGen refers to the fact that the tool has been provided with the *non-packed* goodwares, so that it can avoid strings that are present there as rules features. Results show that yarGen works better when considering its internal strings database (any entry not using the “goodware” parameter), which

Parameters	Cerber	Locky	Upatre	Zerber
rules	18	11	0	10
rules + excludegood	18	11	0	10
rules + goodware	18	11	0	11
rules + excludegood + goodware	18	11	0	11
rules + opcodes	4	4	0	0
rules + opcodes + excludegood	4	4	0	0
rules + opcodes + goodware	4	4	0	11
rules + opcodes + excludegood + goodware	4	4	0	11
rules + z0	3	1	0	7
rules + z0 + excludegood	1	0	0	7
rules + z0 + goodware	5	6	0	1
rules + z0 + excludegood + goodware	2	2	0	0
rules + z0 + opcodes	2	1	0	0
rules + z0 + opcodes + excludegood	1	0	0	0
rules + z0 + opcodes + goodware	3	4	0	1
rules + z0 + opcodes + excludegood + goodware	2	2	0	0

Table 4.6: Table representing the results of the rules produced by yarGen against the packed goodwares. For a complete description of the parameters indicated in each row, it is reminded to [22].

is definitely more complete than the one provided with the goodwares used during these tests, or when taking into account opcodes.
Moreover, even though these kind of statistics will be addressed in the upcoming sections, it should also be mentioned that, for all the family-parameters training combinations there reported, none was capable of fully covering the training set. The highest coverage is, in fact, obtained when the “z0” parameter is set, which relaxes the constraints imposed by the tool when choosing strings that can be used as rules features, and, even in these cases, coverage is never complete.

4.3.3 yaBin results

Interestingly, the yaBin tool, which was stated by the authors to possibly match the packer of the provided samples, have shown promising results, as reported in Table 4.7.
These experiments have been taken by using the only 2 parameters that provided YARA rules for the yaBin tool. What is even more interesting is that, by packing the goodwares, the yaBin false positives numbers, for some of the family drastically decreased (more details on false positives coming in the next sections). Apart from this, as was happening for the yarGen tool, also yaBin was not capable, for some of the families and parameters used, of covering all the samples given as training set.

Parameter	Cerber	Locky	Upatre	Zerber
Yara	0	0	0	0
YaraHunt	0	0	0	0

Table 4.7: Table reporting the results of the false positive matches of yaBin created rules over the families indicated by the respective column and using the parameter in the corresponding row.

4.3.4 Final considerations on the previous results

The just shown results show that YaYaGenPE is somehow resistant to packers detection, at least for the UPX packer. Moreover, the YaYaGenPE tool seems to perform better than yarGen, and equivalently to the yaBin tool. As already remarked, however, this test has not the claim to be absolute, in that it was based on one of the simplest packers openly diffused.

4.4 False Positives matching results

One of the biggest concerns for the cyber-security companies is whether the signatures generated produce false positives or not. In particular, it is requested that signatures (i.e., YARA rules in this work) produce the least possible number of false positive matches.
In order to address this type of issues, it is needed a reliable set of goodwares. Finding a realistic and exhaustive set of goodwares is nowadays a big problem, since the typologies and the variety of executables is really high. In this work, it was used a set of 3,413 PE executables extracted from a Windows 10 Operating System, containing the majority of the softwares the common user has, plus development tools and even some compiled custom (and benign) C programs.
Although it is well known that this might not be enough, for the purposes of a realistic approximation of benign executables used by the community, it seemed the most reasonable choice to come up with, especially considering the lack of a commonly shared and acknowledged goodwares test set.

4.4.1 YaYaGenPE false positives results

Unsupervised Decision Tree Clustering

This section focuses on the results obtained by using the “Unsupervised Decision Tree” clustering algorithm on top of the “clot” or “greedy” algorithms.
Before showing the results, it is needed to mention that, although there are families counting a small number of samples, the clustering algorithm has still been applied. In those cases, if samples are already homogeneous enough, the resulting rules will be exactly

equivalent to those from the plain application of the “greedy” or the “clot” algorithm underneath, as the clustering stops at the first splitting attempt.

Family	**udt**	**udt+rules**	**udt+goodware**	**udt+rules+ goodware**	**udt:clot**	**udt:clot+rules**	**udt:clot+ goodware**	**udt:clot+rules+ goodware**
fraudrop	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
enestaller	0	0	0	0	0	0	0	0
yakes	2	0	0	0	0	0	0	0
hplocky	0	0	0	0	0	0	0	0
teslacrypt	0	0	0	0	0	0	0	0
sagecrypt	0	0	0	0	0	0	0	0
ruskill	0	0	0	0	0	0	0	0
genkryptik	0	0	0	0	0	0	0	0
cerber	2	2	0	0	2	2	0	0
score	0	0	0	0	0	0	0	0
waldek	0	0	0	0	0	0	0	0
dmalocker	0	0	0	0	0	0	0	0
crowti	0	0	0	0	0	0	0	0
zusy	0	0	0	0	0	0	0	0
zbot	0	0	0	0	0	0	0	0
zboter	0	0	0	0	0	0	0	0
allaple	0	0	0	0	0	0	0	0
upatre	0	0	0	0	0	0	0	0
myxah	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
genericcryptor	0	0	0	0	0	0	0	0
glupteba	0	0	0	0	0	0	0	0
locky	0	10	0	0	0	0	0	0
dalexis	0	0	0	0	0	0	0	0
sage	0	0	0	0	0	0	0	0
gamarue	0	0	0	0	0	0	0	0
aura	0	0	0	0	0	0	0	0
cloud	0	0	0	0	0	0	0	0
crypmod	0	0	0	0	0	0	0	0

Continued on next page

Table 4.8 – continued from previous page

Family	**udt**	**udt+rules**	**udt+goodware**	**udt+rules+ goodware**	**udt:clot**	**udt:clot+rules**	**udt:clot+ goodware**	**udt:clot+rules+ goodware**
carberp	0	0	0	0	0	0	0	0
rack	0	0	0	0	0	0	0	0
razy	0	0	0	0	0	0	0	0
coantor	0	0	0	0	0	0	0	0
midie	0	0	0	0	0	0	0	0
tescrypt	0	0	0	0	0	0	0	0
cryptowall	0	0	0	0	0	0	0	0
tinba	0	0	0	0	0	0	0	0
zerber	0	0	0	0	0	0	0	0
delf	0	0	0	0	0	0	0	0
teerac	0	0	0	0	0	0	0	0
enestedel	0	0	0	0	0	0	0	0
mikey	0	0	0	0	0	0	0	0
bitman	0	0	0	0	0	0	0	0
barys	0	0	0	0	0	0	0	0
scatter	0	0	0	0	0	0	0	0
dridex	0	0	0	0	0	0	0	0
shiz	0	0	0	0	0	0	0	0
agentb	0	0	0	0	0	0	0	0
lethic	0	0	0	0	0	0	0	0
shade	0	0	0	0	0	0	0	0
reconyc	0	0	0	0	0	0	0	0
deshacop	0	0	0	0	0	0	0	0
fareit	0	0	0	0	0	0	0	0
dynamer	0	0	0	0	0	0	0	0
torrentlocker	0	0	0	0	0	0	0	0
onion	0	0	0	0	0	0	0	0
critroni	0	0	0	0	0	0	0	0
dagozill	0	0	0	0	0	0	0	0
tpyn	0	0	0	0	0	0	0	0
zegost	0	0	0	0	0	0	0	0
ngrbot	0	0	0	0	0	0	0	0

Continued on next page

Table 4.8 – continued from previous page

Family	**udt**	**udt+rules**	**udt+goodware**	**udt+rules+ goodware**	**udt:clot**	**udt:clot+rules**	**udt:clot+ goodware**	**udt:clot+rules+ goodware**
scar	0	0	0	0	0	0	0	0
beebone	0	0	0	0	0	0	0	0

Table 4.8: Table that represents the number of false positives found by the rules trained with algorithms and parameters indicated in the respective column using, as training set, the family indicated in the respective row. The rules have been generated using the "Russell Rao" distance. Whenever not explicitly specified the "clot" algorithm, it is implied the usage of the "greedy" to determine the final rules.

Table 4.8 show the number of False Positives found by the rules generated using all the possible parameters combinations over all the families detected by AVclass. All the trainings have been done by using the "Russell Rao" distance.
Results show no False Positive matches for almost all the families and parameters combinations, and, although these numbers may vary slightly for the biggest families, the number of matched False positives is always close to those already shown in Table 4.8.
Similar results were obtained by performing the same set of trainings and tests but using the "Jaccard" distance instead of the "Russell Rao" one, as shown in Table 4.9.

Family	**udt**	**udt+rules**	**udt+goodware**	**udt+rules+ goodware**	**udt:clot**	**udt:clot+rules**	**udt:clot+ goodware**	**udt:clot+rules+ goodware**
fraudrop	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
enestaller	0	0	0	0	0	0	0	0
yakes	0	0	0	0	0	0	0	0
hplocky	0	0	0	0	0	0	0	0
teslacrypt	0	0	0	0	0	0	0	0
sagecrypt	0	0	0	0	0	0	0	0
ruskill	0	0	0	0	0	0	0	0

Continued on next page

Table 4.9 – continued from previous page

Family	udt	udt+rules	udt+goodware	udt+rules+ goodware	udt:clot	udt:clot+rules	udt:clot+ goodware	udt:clot+rules+ goodware
genkryptik	0	0	0	0	0	0	0	0
cerber	0	0	0	0	0	0	0	0
score	0	0	0	0	0	0	0	0
waldek	0	0	0	0	0	0	0	0
dmalocker	0	0	0	0	0	0	0	0
crowti	0	0	0	0	0	0	0	0
zusy	0	0	0	0	0	0	0	0
zbot	0	0	0	0	0	0	0	0
zboter	0	0	0	0	0	0	0	0
allaple	0	0	0	0	0	0	0	0
upatre	0	0	0	0	0	0	0	0
myxah	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
genericcryptor	0	0	0	0	0	0	0	0
glupteba	0	0	0	0	0	0	0	0
locky	10	0	0	0	1	0	0	0
dalexis	0	0	0	0	0	0	0	0
sage	0	0	0	0	0	0	0	0
gamarue	0	0	0	0	0	0	0	0
aura	0	0	0	0	0	0	0	0
cloud	0	0	0	0	0	0	0	0
crypmod	0	0	0	0	0	0	0	0
carberp	0	0	0	0	0	0	0	0
rack	0	0	0	0	0	0	0	0
razy	0	0	0	0	0	0	0	0
coantor	0	0	0	0	0	0	0	0
midie	0	0	0	0	0	0	0	0
tescrypt	0	0	0	0	0	0	0	0
cryptowall	0	0	0	0	0	0	0	0
tinba	0	0	0	0	0	0	0	0
zerber	0	0	0	0	0	0	0	0
delf	0	0	0	0	0	0	0	0

Continued on next page

Table 4.9 – continued from previous page

Family	udt	udt+rules	udt+goodware	udt+rules+ goodware	udt:clot	udt:clot+rules	udt:clot+ goodware	udt:clot+rules+ goodware
teerac	0	0	0	0	0	0	0	0
enestedel	0	0	0	0	0	0	0	0
mikey	0	0	0	0	0	0	0	0
bitman	0	0	0	0	0	0	0	0
barys	0	0	0	0	0	0	0	0
scatter	0	0	0	0	0	0	0	0
dridex	0	0	0	0	0	0	0	0
shiz	0	0	0	0	0	0	0	0
agentb	0	0	0	0	0	0	0	0
lethic	0	0	0	0	0	0	0	0
shade	0	0	0	0	0	0	0	0
reconyc	0	0	0	0	0	0	0	0
deshacop	0	0	0	0	0	0	0	0
fareit	0	0	0	0	0	0	0	0
dynamer	0	0	0	0	0	0	0	0
torrentlocker	0	0	0	0	0	0	0	0
onion	0	0	0	0	0	0	0	0
critroni	0	0	0	0	0	0	0	0
dagozill	0	0	0	0	0	0	0	0
tpyn	0	0	0	0	0	0	0	0
zegost	0	0	0	0	0	0	0	0
ngrbot	0	0	0	0	0	0	0	0
scar	0	0	0	0	0	0	0	0
beebone	0	0	0	0	0	0	0	0

Table 4.9: Table that represents the number of false positives found by the rules trained with algorithms and parameters indicated in the respective column using, as training set, the family indicated in the respective row. The rules have been generated using the “Jaccard” distance. Whenever not explicitly specified the “clot” algorithm, it is implied the usage of the “greedy” to determine the final rules.

HDBSCAN clustering algorithm

In this section it will be presented the same set of results just shown using the “Unsupervised Decision Tree”, but, this time, using the “HDBSCAN” clustering.
The results are slightly higher, in terms of False positives, with respect to the corresponding ones using the “Unsupervised Decision Tree”, but still, they present a small number of False Positives for all the families.

Family	**hdbscan**	**hdbscan+rules**	**hdbscan+goodware**	**hdbscan+rules+ goodware**	**hdbscan:clot**	**hdbscan:clot+rules**	**hdbscan:clot+ goodware**	**hdbscan:clot+rules+ goodware**
fraudrop	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
enestaller	0	0	0	0	0	0	0	0
yakes	0	0	0	0	0	0	0	0
hplocky	0	0	0	0	0	0	0	0
teslacrypt	0	0	0	0	0	0	0	0
sagecrypt	0	0	0	0	0	0	0	0
ruskill	0	0	0	0	0	0	0	0
genkryptik	0	0	0	0	0	0	0	0
cerber	0	0	0	0	0	0	0	0
score	0	0	0	0	0	0	0	0
waldek	0	0	0	0	0	0	0	0
dmalocker	0	0	0	0	0	0	0	0
crowti	1	0	0	0	0	0	0	0
zusy	0	0	0	0	0	0	0	0
zbot	0	0	0	0	0	0	0	0
zboter	0	0	0	0	0	0	0	0
allaple	0	0	0	0	0	0	0	0
upatre	2	0	0	0	1	0	0	0
myxah	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
genericcryptor	0	0	0	0	0	0	0	0
glupteba	0	0	0	0	0	0	0	0
locky	7	1	0	0	4	1	0	0

Continued on next page

Table 4.10 – continued from previous page

Family	**hdbscan**	**hdbscan+rules**	**hdbscan+goodware**	**hdbscan+rules+ goodware**	**hdbscan:clot**	**hdbscan:clot+rules**	**hdbscan:clot+ goodware**	**hdbscan:clot+rules+ goodware**
dalexis	0	0	0	0	0	0	0	0
sage	0	0	0	0	0	0	0	0
gamarue	0	0	0	0	0	0	0	0
aura	0	0	0	0	0	0	0	0
cloud	0	0	0	0	0	0	0	0
crypmod	0	0	0	0	0	0	0	0
carberp	0	0	0	0	0	0	0	0
rack	0	0	0	0	0	0	0	0
razy	0	0	0	0	0	0	0	0
coantor	0	0	0	0	0	0	0	0
midie	0	0	0	0	0	0	0	0
tescrypt	0	0	0	0	0	0	0	0
cryptowall	0	0	0	0	0	0	0	0
tinba	0	0	0	0	0	0	0	0
zerber	2	0	0	0	2	0	0	0
delf	0	0	0	0	0	0	0	0
teerac	0	0	0	0	0	0	0	0
enestedel	0	0	0	0	0	0	0	0
mikey	0	0	0	0	0	0	0	0
bitman	0	0	0	0	0	0	0	0
barys	0	0	0	0	0	0	0	0
scatter	5	0	0	0	5	0	0	0
dridex	0	0	0	0	0	0	0	0
shiz	0	0	0	0	0	0	0	0
agentb	0	0	0	0	0	0	0	0
lethic	0	0	0	0	0	0	0	0
shade	0	0	0	0	0	0	0	0
reconyc	0	0	0	0	0	0	0	0
deshacop	0	0	0	0	0	0	0	0
fareit	0	0	0	0	0	0	0	0

Continued on next page

Table 4.10 – continued from previous page

Family	**hdbscan**	**hdbscan+rules**	**hdbscan+goodware**	**hdbscan+rules+ goodware**	**hdbscan:clot**	**hdbscan:clot+rules**	**hdbscan:clot+ goodware**	**hdbscan:clot+rules+ goodware**
dynamer	0	0	0	0	0	0	0	0
torrentlocker	0	0	0	0	0	0	0	0
onion	8	1	0	0	0	0	0	0
critroni	0	0	0	0	0	0	0	0
dagozill	0	0	0	0	0	0	0	0
tpyn	0	0	0	0	0	0	0	0
zegost	0	0	0	0	0	0	0	0
ngrbot	0	0	0	0	0	0	0	0
scar	0	0	0	0	0	0	0	0
beebone	0	0	0	0	0	0	0	0

Table 4.10: Table that represents the number of false positives found by the rules trained with algorithms and parameters indicated in the respective column using, as training set, the family indicated in the respective row. The rules have been generated using the "Russell Rao" distance. Whenever not explicitly specified the "clot" algorithm, it is implied the usage of the "greedy" to determine the final rules.

Table 4.10 shows a behavior that is also evidenced in the majority of the families presenting false positives in Table 4.8, Table 4.9 and Table 4.11. In particular, it is possible to see that the addition of the user provided rules as features tends, in several cases, to decrease the number of false positives matched, sometimes also drastically. This behavior is somewhat expected, in that, user provided rules, might target strings and opcodes that successfully distinguish malicious executables from benign one. Specifically, during all the tests performed, it has been made use of the rules cited in subsection 4.1.3, which, although not perfect, contain a huge number of packers and specific opcodes matching rules.
These rules might be the reason why the number of false positives, in most of the false positives matching rules, decreases, in that, most of benign samples are not packed and are compiled through standard compilers, which tend to have a more regular set of opcodes (i.e., it is unlikely that they will contain the opcodes searched by these rules).

Family	**hdbscan**	**hdbscan+rules**	**hdbscan+goodware**	**hdbscan+rules+ goodware**	**hdbscan:clot**	**hdbscan:clot+rules**	**hdbscan:clot+ goodware**	**hdbscan:clot+rules+ goodware**
fraudrop	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
enestaller	0	0	0	0	0	0	0	0
yakes	0	0	0	0	0	0	0	0
hplocky	0	0	0	0	0	0	0	0
teslacrypt	0	0	0	0	0	0	0	0
sagecrypt	0	0	0	0	0	0	0	0
ruskill	0	0	0	0	0	0	0	0
genkryptik	0	0	0	0	0	0	0	0
cerber	0	0	0	0	0	0	0	0
score	0	0	0	0	0	0	0	0
waldek	0	0	0	0	0	0	0	0
dmalocker	0	0	0	0	0	0	0	0
crowti	6	9	0	0	16	9	0	0
zusy	0	0	0	0	0	0	0	0
zbot	0	0	0	0	0	0	0	0
zboter	0	0	0	0	0	0	0	0
allaple	0	0	0	0	0	0	0	0
upatre	1	2	0	0	1	2	0	0
myxah	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
genericcryptor	0	0	0	0	0	0	0	0
glupteba	0	0	0	0	0	0	0	0
locky	0	0	0	0	0	1	0	0
dalexis	0	0	0	0	0	0	0	0
sage	0	0	0	0	0	0	0	0
gamarue	0	0	0	0	0	0	0	0
aura	0	0	0	0	0	0	0	0
cloud	0	0	0	0	0	0	0	0
crypmod	0	0	0	0	0	0	0	0

Continued on next page

Table 4.11 – continued from previous page

Family	**hdbscan**	**hdbscan+rules**	**hdbscan+goodware**	**hdbscan+rules+ goodware**	**hdbscan:clot**	**hdbscan:clot+rules**	**hdbscan:clot+ goodware**	**hdbscan:clot+rules+ goodware**
carberp	0	0	0	0	0	0	0	0
rack	0	0	0	0	0	0	0	0
razy	0	0	0	0	0	0	0	0
coantor	0	0	0	0	0	0	0	0
midie	0	0	0	0	0	0	0	0
tescrypt	11	7	0	0	11	7	0	0
cryptowall	2	0	0	0	8	0	0	0
tinba	0	0	0	0	0	0	0	0
zerber	0	0	0	0	0	0	0	0
delf	0	0	0	0	0	0	0	0
teerac	0	0	0	0	0	0	0	0
enestedel	0	0	0	0	0	0	0	0
mikey	0	0	0	0	0	0	0	0
bitman	0	0	0	0	0	0	0	0
barys	0	0	0	0	0	0	0	0
scatter	0	0	0	0	0	0	0	0
dridex	0	0	0	0	0	0	0	0
shiz	0	0	0	0	0	0	0	0
agentb	0	0	0	0	0	0	0	0
lethic	0	0	0	0	0	0	0	0
shade	0	0	0	0	0	0	0	0
reconyc	0	0	0	0	0	0	0	0
deshacop	0	0	0	0	0	0	0	0
fareit	0	0	0	0	0	0	0	0
dynamer	0	0	0	0	0	0	0	0
torrentlocker	0	0	0	0	0	0	0	0
onion	0	0	0	0	0	0	0	0
critroni	0	0	0	0	0	0	0	0
dagozill	0	0	0	0	0	0	0	0
tpyn	0	0	0	0	0	0	0	0

Continued on next page

Table 4.11 – continued from previous page

Family	hdbscan	hdbscan+rules	hdbscan+goodware	hdbscan+rules+ goodware	hdbscan:clot	hdbscan:clot+rules	hdbscan:clot+ goodware	hdbscan:clot+rules+ goodware
zegost	0	0	0	0	0	0	0	0
ngrbot	0	0	0	0	0	0	0	0
scar	0	0	0	0	0	0	0	0
beebone	0	0	0	0	0	0	0	0

Table 4.11: Table that represents the number of false positives found by the rules trained with algorithms and parameters indicated in the respective column using, as training set, the family indicated in the respective row. The rules have been generated using the "Jaccard" distance. Whenever not explicitly specified the "clot" algorithm, it is implied the usage of the "greedy" to determine the final rules.

4.4.2 yarGen False Positives statistics

The yarGen tool disposes of a lot of possible parameters and their combinations, however, when considering medium-large families, the majority of them present a non-zero, and sometimes also relevant, number of false positives. The resulting number of false positives found for the rules produced by yarGen are in Table 4.12 and Table 4.13, where they present the same usage of parameters per training, except for the parameter "z0" which was used exclusively in Table 4.13. The only difference introduced by this parameter is the fact that the scoring associated to each string is ignored when adding a string to the generated rules, i.e., any valuable string found is added to the rule, independently on how the tool evaluates it. This behavior usually results in a better coverage of the training set, however, it might be a sort of double-edged sword, in that, for some families, it introduces a spike in the number of false positives.

Family	**plain**	**goodware**	**excludegood**	**goodware+ excludegood**	**opcodes**	**opcodes+ goodware**	**opcodes+ excludegood**	**opcodes+goodware+ excludegood**
fraudrop	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
enestaller	0	0	0	0	0	0	0	0
yakes	0	0	0	0	0	0	0	0
hplocky	0	0	0	0	0	0	0	0
teslacrypt	0	0	0	0	0	0	0	0
sagecrypt	0	0	0	0	0	0	0	0
ruskill	0	0	0	0	0	0	0	0
genkryptik	0	0	0	0	0	0	0	0
cerber	19	20	19	20	7	4	7	4
score	0	0	0	0	0	0	0	0
waldek	0	0	0	0	0	0	0	0
dmalocker	0	0	0	0	0	0	0	0
crowti	1	2	1	2	0	0	0	0
zusy	0	1	0	1	0	0	0	0
zbot	0	1	0	1	0	0	0	0
zboter	0	0	0	0	0	0	0	0
allaple	0	0	0	0	0	0	0	0
upatre	2	2	2	2	0	0	0	0
myxah	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
genericcryptor	0	0	0	0	0	0	0	0
glupteba	0	0	0	0	0	0	0	0
locky	25	22	25	22	7	4	7	4
dalexis	0	0	0	0	0	0	0	0
sage	0	0	0	0	0	0	0	0
gamarue	0	0	0	0	0	0	0	0
aura	0	0	0	0	0	0	0	0
cloud	0	0	0	0	0	0	0	0
crypmod	3	0	3	0	0	0	0	0

Continued on next page

Table 4.12 – continued from previous page

Family	**plain**	**goodware**	**excludegood**	**goodware+ excludegood**	**opcodes**	**opcodes+ goodware**	**opcodes+ excludegood**	**opcodes+goodware+ excludegood**
carberp	0	0	0	0	0	0	0	0
rack	0	0	0	0	0	0	0	0
razy	0	0	0	0	0	0	0	0
coantor	0	0	0	0	0	0	0	0
midie	0	0	0	0	0	0	0	0
tescrypt	0	0	0	0	0	0	0	0
cryptowall	15	1	15	1	0	0	0	0
tinba	0	1	0	1	0	0	0	0
zerber	28	19	28	19	6	19	6	19
delf	0	0	0	0	0	0	0	0
teerac	0	0	0	0	0	0	0	0
enestedel	0	0	0	0	0	0	0	0
mikey	0	0	0	0	0	0	0	0
bitman	21	5	21	5	0	0	0	0
barys	0	0	0	0	0	0	0	0
scatter	6	19	6	19	6	19	6	19
dridex	0	0	0	0	0	0	0	0
shiz	0	0	0	0	0	0	0	0
agentb	0	0	0	0	0	0	0	0
lethic	0	1	0	1	0	0	0	0
shade	6	19	6	19	6	19	6	19
reconyc	0	0	0	0	0	0	0	0
deshacop	0	0	0	0	0	0	0	0
fareit	0	0	0	0	0	0	0	0
dynamer	6	4	6	4	6	4	6	4
torrentlocker	0	0	0	0	0	0	0	0
onion	0	0	0	0	0	0	0	0
critroni	0	0	0	0	0	0	0	0
dagozill	0	0	0	0	0	0	0	0
tpyn	0	0	0	0	0	0	0	0

Continued on next page

Table 4.12 – continued from previous page

Family	**plain**	**goodware**	**excludegood**	**goodware+ excludegood**	**opcodes**	**opcodes+ goodware**	**opcodes+ excludegood**	**opcodes+goodware+ excludegood**
zegost	0	0	0	0	0	0	0	0
ngrbot	0	0	0	0	0	0	0	0
scar	0	0	0	0	0	0	0	0
beebone	0	0	0	0	0	0	0	0

Table 4.12: Table that represents the number of false positives found by the rules generated by yarGen with the parameters indicated in the respective column using, as training set, the family indicated in the respective row.

Family	**plain**	**goodware**	**excludegood**	**goodware+ excludegood**	**opcodes**	**opcodes+ goodware**	**opcodes+ excludegood**	**opcodes+ goodware+ excludegood**
fraudrop	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
enestaller	0	0	0	0	0	0	0	0
yakes	295	295	0	0	0	0	0	0
hplocky	0	0	0	0	0	0	0	0
teslacrypt	2	2	0	0	0	0	0	0
sagecrypt	295	295	0	0	0	0	0	0
ruskill	0	0	0	0	0	0	0	0
genkryptik	0	0	0	0	0	0	0	0
cerber	7	9	4	2	6	3	4	2
score	0	0	0	0	0	0	0	0
waldek	0	0	0	0	0	0	0	0
dmalocker	0	0	0	0	0	0	0	0
crowti	0	0	0	0	0	0	0	0
zusy	2	2	0	0	0	0	0	0
zbot	0	0	0	0	0	0	0	0

Continued on next page

Table 4.13 – continued from previous page

Family	plain	goodware	excludegood	goodware+ excludegood	opcodes	opcodes+ goodware	opcodes+ excludegood	opcodes+ goodware+ excludegood
zboter	0	0	0	0	0	0	0	0
allaple	0	0	0	0	0	0	0	0
upatre	2	6	2	0	0	0	0	0
myxah	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
genericcryptor	0	0	0	0	0	0	0	0
glupteba	0	0	0	0	0	0	0	0
locky	6	14	4	3	5	4	4	2
dalexis	0	0	0	0	0	0	0	0
sage	0	0	0	0	0	0	0	0
gamarue	0	3	0	0	0	1	0	0
aura	0	0	0	0	0	0	0	0
cloud	0	0	0	0	0	0	0	0
crypmod	0	0	0	0	0	0	0	0
carberp	0	0	0	0	0	0	0	0
rack	0	0	0	0	0	0	0	0
razy	0	0	0	0	0	0	0	0
coantor	0	0	0	0	0	0	0	0
midie	0	0	0	0	0	0	0	0
tescrypt	0	0	0	0	0	0	0	0
cryptowall	1	4	0	0	0	0	0	0
tinba	0	1	0	0	0	0	0	0
zerber	25	1	25	0	4	1	4	0
delf	0	0	0	0	0	0	0	0
teerac	0	0	0	0	0	0	0	0
enestedel	0	0	0	0	0	0	0	0
mikey	0	0	0	0	0	0	0	0
bitman	3	17	0	1	0	0	0	0
barys	0	0	0	0	0	0	0	0
scatter	5	6	4	2	4	5	4	2
dridex	0	0	0	0	0	0	0	0
shiz	0	0	0	0	0	0	0	0
agentb	0	0	0	0	0	0	0	0

Continued on next page

Table 4.13 – continued from previous page

Family	plain	goodware	excludegood	goodware+ excludegood	opcodes	opcodes+ goodware	opcodes+ excludegood	opcodes+ goodware+ excludegood
lethic	0	0	0	0	0	0	0	0
shade	4	4	4	0	4	4	4	0
reconyc	0	0	0	0	0	0	0	0
deshacop	0	0	0	0	0	0	0	0
fareit	0	0	0	0	0	0	0	0
dynamer	4	5	4	2	4	5	4	2
torrentlocker	0	0	0	0	0	0	0	0
onion	1	2	0	0	0	0	0	0
critroni	0	0	0	0	0	0	0	0
dagozill	0	0	0	0	0	0	0	0
tpyn	0	0	0	0	0	0	0	0
zegost	0	0	0	0	0	0	0	0
ngrbot	0	0	0	0	0	0	0	0
scar	0	0	0	0	0	0	0	0
beebone	0	0	0	0	0	0	0	0

Table 4.13: Table that represents the number of false positives found by the rules generated by yarGen with the parameters indicated in the respective column using, as training set, the family indicated in the respective row. The parameter "z0", although not shown explicitly in the table was permanently set during all the trainings performed.

As results show, yarGen rules tend to generate false positives in several tests, particularly when considering families with a numerous number of samples, according to the distribution in Figure 4.1. In particular, the number of False Positives found by yarGen is much higher than the number of False Positives found by YaYaGenPE.

4.4.3 yaBin false positives

yaBin shows alternating results in terms of false positives, depending on the parameter it is being used. Results in Table 4.14 show that the usage of the "yara" parameter, which generates really specific YARA rules, i.e., rules searching for several function prologues each, present no false positives in all the tested families. On the other hand, the parameter "yaraHunt", which tries to simplify the rules generated by the "yara" parameter in order to match several different samples, are affected by an high number of false positives. This compromise was already stated by the authors, and, although it creates a large number of false positives, it successfully increases the training set coverage by the generated rules, which, otherwise, is not guaranteed to be complete.

Family	**yara**	**yaraHunt**
fraudrop	0	5
enestaller	0	0
yakes	0	77
teslacrypt	0	68
sagecrypt	0	52
genkryptik	0	0
cerber	0	16
waldek	0	0
crowti	0	6
zusy	0	12
zbot	0	5
allaple	0	0
upatre	0	20
glupteba	0	52
locky	0	57
dalexis	0	1
sage	0	3
gamarue	0	1
crypmod	0	9
rack	0	0
razy	0	3
midie	0	1
tescrypt	0	5
cryptowall	0	10
tinba	0	1
zerber	0	12
teerac	0	0
enestedel	0	0
mikey	0	1
bitman	0	32
barys	0	0
scatter	0	2
dridex	0	2
shiz	0	1
agentb	0	2

Continued on next page

Table 4.14 – continued from previous page

Family	yara	yaraHunt
lethic	0	0
reconyc	0	2
deshacop	0	1
fareit	0	6
dynamer	0	0
onion	0	3
critroni	0	0
dagozill	0	0
tpyn	0	2
scar	0	1
beebone	0	0

Table 4.14: Table that represents the number of false positives found by the rules generated by yaBin with the parameters indicated in the respective column using, as training set, the family indicated in the respective row.

Comparing the just presented results with those of YaYaGenPE, it is evidenced that yaBin tends to perform slightly better than YaYaGenPE when using the “yara” parameter, while definitely worse with the “yaraHunt” one.

4.5 True positives and malware families coverage testing

In this section, it is addressed the problem of how well do the rules detect malicious samples they weren’t purposely written for. This test has to be compared with the previous one, in that, although it is preferred to have rules that do not match any goodware at all, it is as well desirable that the rules are capable of detecting samples they weren’t trained on, since, otherwise, it would be probably simpler to address the malwares’ hashes as rule conditions.
The tests have been performed by generating the rules for each family in the dataset and applying them on the remaining part of the executables. Indeed, although different families should have different distinctive traits, detecting the real family of an executable is difficult. This means that, using the AVclass assigned labels to each sample, it is likely that some malwares are mis-classified, especially when considering executables that are ambiguous enough to be catergorized by an equivalent number of Anti-Viruses as belonging to two or more different families. That might happen due to the not perfect AVclass aliases removal or because the sample is actually classified differently by the Anti-Viruses. The bottom line is that some families used for the training might contain samples not belonging to

that family and, some executables not in the training set might effectively be samples of the training family.

4.5.1 YaYaGenPE true positives

Unsupervised Decision Tree results

Table 4.15 and Table 4.16 show the number of new malware positives found by the rules generated on the families indicated in the rows, using the parameters indicated in the columns and, respectively, the Russell Rao and Jaccard distances. The amount of new positives is determined by applying the generated rules on the remaining part of the dataset. This set of results give an idea of how well do the rules target novel and possibly similar malicious samples.

Family	**udt**	**udt+rules**	**udt+goodware**	**udt+rules+ goodware**	**udt:clot**	**udt:clot+rules**	**udt:clot+ goodware**	**udt:clot+rules+ goodware**
agentb	1	1	1	1	1	1	1	1
allaple	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
aura	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
barys	3	3	3	3	3	3	3	3
beebone	1	1	1	1	1	1	1	1
bitman	977	984	895	1010	957	1003	923	1030
carberp	0	0	0	0	0	0	0	0
cerber	321	363	403	358	305	344	306	349
cloud	0	0	0	0	0	0	0	0
coantor	0	0	0	0	0	0	0	0
critroni	0	0	0	0	0	0	0	0
crowti	66	61	62	62	62	61	58	58
crypmod	92	79	97	30	84	30	128	79
cryptowall	25	29	25	29	29	25	29	26
dagozill	1	4	1	4	1	1	1	1
dalexis	2	2	2	2	2	2	2	2
delf	0	0	0	0	0	0	0	0
deshacop	0	0	0	0	0	0	0	0

Continued on next page

Table 4.15 – continued from previous page

Family	udt	udt+rules	udt+goodware	udt+rules+ goodware	udt:clot	udt:clot+rules	udt:clot+ goodware	udt:clot+rules+ goodware
dmalocker	0	0	0	0	0	0	0	0
dridex	0	0	0	0	0	0	0	0
dynamer	5	5	5	5	5	5	5	5
enestaller	9	9	9	9	9	9	9	9
enestedel	100	100	100	100	100	100	100	100
fareit	2	2	2	2	2	2	2	2
fraudrop	3	3	0	3	3	3	3	3
gamarue	11	5	11	5	15	6	12	7
genericcryptor	1	1	1	1	1	1	1	1
genkryptik	4	4	4	4	4	4	4	4
glupteba	44	44	44	44	44	44	44	44
hplocky	3	3	3	3	3	3	3	3
lethic	12	12	12	12	12	12	12	12
locky	28	168	116	37	114	121	111	36
midie	1	1	1	1	1	1	1	1
mikey	7	14	7	7	7	14	7	14
myxah	13	13	13	13	13	13	13	13
ngrbot	0	0	0	0	0	0	0	0
onion	14	12	14	13	14	12	14	13
rack	0	0	0	0	0	0	0	0
razy	78	95	78	94	78	87	143	95
reconyc	0	0	0	0	0	0	0	0
ruskill	0	0	0	0	0	0	0	0
sage	13	13	13	13	13	13	13	13
sagecrypt	29	43	30	43	29	31	30	43
scar	11	11	11	11	11	11	11	11
scatter	11	11	11	11	11	11	11	11
score	0	0	0	0	0	0	0	0
shade	0	0	0	0	0	0	0	0
shiz	105	62	99	68	102	102	96	67
teerac	0	0	0	0	0	0	0	0
tescrypt	60	54	60	60	60	60	71	60

Continued on next page

Table 4.15 – continued from previous page

Family	**udt**	**udt+rules**	**udt+goodware**	**udt+rules+ goodware**	**udt:clot**	**udt:clot+rules**	**udt:clot+ goodware**	**udt:clot+rules+ goodware**
teslacrypt	867	885	897	874	889	884	866	904
tinba	9	9	9	9	9	9	9	9
torrentlocker	1	1	1	1	1	1	1	1
tpyn	1	1	1	1	1	1	1	1
upatre	21	34	27	22	26	28	25	29
waldek	0	0	0	0	0	0	0	0
yakes	103	105	95	101	95	97	95	109
zbot	4	4	4	4	4	4	4	4
zboter	0	0	0	0	0	0	0	0
zegost	0	0	0	0	0	0	0	0
zerber	493	454	471	483	439	488	457	445
zusy	63	97	78	97	78	69	65	97

Table 4.15: Table that shows the number of new malware positives found by the rules trained on the family on the row with the parameters indicated by the respective column. All the rules used for the tests have been generated using the Russell Rao distance.

Family	**udt**	**udt+rules**	**udt+goodware**	**udt+rules+ goodware**	**udt:clot**	**udt:clot+rules**	**udt:clot+ goodware**	**udt:clot+rules+ goodware**
agentb	1	1	1	1	1	1	1	1
allaple	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
aura	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
barys	3	3	3	3	3	3	3	3
beebone	1	1	1	1	1	1	1	1
bitman	961	1051	901	937	898	1003	944	916

Continued on next page

Table 4.16 – continued from previous page

Family	udt	udt+rules	udt+goodware	udt+rules+ goodware	udt:clot	udt:clot+rules	udt:clot+ goodware	udt:clot+rules+ goodware
carberp	0	0	0	0	0	0	0	0
cerber	338	336	417	355	323	303	331	352
cloud	0	0	0	0	0	0	0	0
coantor	0	0	0	0	0	0	0	0
critroni	0	0	0	0	0	0	0	0
crowti	62	58	62	62	58	58	57	58
crypmod	104	79	84	79	127	79	119	30
cryptowall	29	26	28	25	29	25	29	25
dagozill	1	4	1	4	1	1	1	1
dalexis	2	2	2	2	2	2	2	2
delf	0	0	0	0	0	0	0	0
deshacop	0	0	0	0	0	0	0	0
dmalocker	0	0	0	0	0	0	0	0
dridex	0	0	0	0	0	0	0	0
dynamer	5	5	5	5	5	5	5	5
enestaller	9	9	9	9	9	9	9	9
enestedel	100	100	100	100	100	100	100	100
fareit	2	2	2	2	2	2	2	1
fraudrop	3	3	3	3	3	3	0	0
gamarue	12	4	6	8	12	4	14	4
genericcryptor	1	1	1	1	1	1	1	1
genkryptik	4	4	4	4	4	4	4	4
glupteba	44	44	44	44	44	44	44	44
hplocky	3	3	3	3	3	3	3	3
lethic	12	12	12	12	12	12	12	12
locky	303	110	110	35	263	42	114	116
midie	1	1	1	1	1	1	1	1
mikey	7	7	7	14	7	14	7	7
myxah	13	13	13	13	13	13	13	13
ngrbot	0	0	0	0	0	0	0	0
onion	14	12	14	13	14	12	14	13
rack	0	0	0	0	0	0	0	0

Continued on next page

Table 4.16 – continued from previous page

Family	**udt**	**udt+rules**	**udt+goodware**	**udt+rules+ goodware**	**udt:clot**	**udt:clot+rules**	**udt:clot+ goodware**	**udt:clot+rules+ goodware**
razy	96	144	87	278	87	88	96	95
reconyc	0	0	0	0	0	0	0	0
ruskill	0	0	0	0	0	0	0	0
sage	13	13	13	13	13	13	13	13
sagecrypt	35	43	35	43	30	43	35	43
scar	11	11	11	8	11	11	11	11
scatter	11	11	11	11	11	11	11	11
score	0	0	0	0	0	0	0	0
shade	0	0	0	0	0	0	0	0
shiz	79	67	103	67	99	67	87	67
teerac	0	0	0	0	0	0	0	0
tescrypt	60	60	71	60	85	85	85	60
teslacrypt	871	895	898	860	878	908	839	925
tinba	9	9	9	9	9	9	9	9
torrentlocker	1	1	1	1	1	1	1	1
tpyn	1	1	1	1	1	1	1	1
upatre	32	33	23	13	22	34	23	32
waldek	0	0	0	0	0	0	0	0
yakes	112	99	111	109	96	102	103	114
zbot	4	4	4	4	4	4	4	4
zboter	0	0	0	0	0	0	0	0
zegost	0	0	0	0	0	0	0	0
zerber	447	479	493	425	459	441	404	411
zusy	72	91	65	97	63	97	65	93

Table 4.16: Table that shows the number of new malware positives found by the rules trained on the family on the row with the parameters indicated by the respective column. All the rules used for the tests have been generated using the Jaccard distance.

Results in the tables are comparable in terms of numbers and they both show that the rules are generic enough to match other samples in the dataset. Specifically, taking a look at Figure 4.1, it is possible to verify that the biggest families are also those that tend to be the best ones in terms of new positives. This is might be consequence of two

particular reasons: the number of samples inside each family and the imprecise AVclass family detection. The former implies that, to cover a huge number of samples, it is required an adequately large set of rules, which, if generic enough, are capable of covering several other malicious executables. The latter suggests that, when using AVclass labels to identify the families, it is more likely that, on bigger families, there might be mis-classified samples. This mis-classification leads to some rules that, in fact, target samples belonging to different families with respect to the training one, and, as a consequence, they are likely to match other samples of those families.
Finally, some of the biggest families are also those for which aliases were not perfectly detected, since some of these are actually aliases of some others (e.g., “teslacrypt” and “bitman” have been reported to be the same family for what concerns Kaspersky Labs, while “cerber” and “zerber” should be variants of the same family). For this reason, manually verifying the results evidenced that rules generated on each family is capable of covering several other samples of the respective alias family.

4.5.2 HDBSCAN results

This section presents the same set of results as the previous one, but determined using the rules found on the clusters determined by the HDBSCAN algorithm. All the results have been grouped in two tables: Table 4.17, which reports the results of the rules created using the Russell Rao distance, and Table 4.18, that presents results of the rules determined using the Jaccard distance.

Family	**hdbscan**	**hdbscan+rules**	**hdbscan+goodware**	**hdbscan+rules+ goodware**	**hdbscan:clot**	**hdbscan:clot+rules**	**hdbscan:clot+ goodware**	**hdbscan:clot+rules+ goodware**
agentb	1	1	1	1	1	1	1	1
allaple	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
aura	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
barys	3	3	3	3	3	3	3	3
beebone	1	1	1	1	1	1	1	1
bitman	1681	1679	1681	1679	1680	1679	1680	1678
carberp	0	0	0	0	0	0	0	0

Continued on next page

Table 4.17 – continued from previous page

Family	hdbscan	hdbscan+rules	hdbscan+goodware	hdbscan+rules+ goodware	hdbscan:clot	hdbscan:clot+rules	hdbscan:clot+ goodware	hdbscan:clot+rules+ goodware
cerber	297	293	297	293	251	247	251	247
cloud	0	0	0	0	0	0	0	0
coantor	0	0	0	0	0	0	0	0
critroni	0	0	0	0	0	0	0	0
crowti	79	79	66	79	61	61	61	61
crypmod	122	122	122	122	122	122	122	122
cryptowall	41	40	41	40	41	40	41	40
dagozill	4	4	4	4	1	1	1	1
dalexis	3	3	3	3	3	3	3	3
delf	0	0	0	0	0	0	0	0
deshacop	1	1	1	1	1	1	1	1
dmalocker	0	0	0	0	0	0	0	0
dridex	0	0	0	0	0	0	0	0
dynamer	5	5	5	5	5	5	5	5
enestaller	9	9	9	9	9	9	9	9
enestedel	4	4	4	4	4	4	4	4
fareit	2	2	2	2	2	2	2	2
fraudrop	0	0	0	0	0	0	0	0
gamarue	18	4	18	18	6	6	19	18
genericcryptor	1	1	1	1	1	1	1	1
genkryptik	4	4	4	4	4	4	4	4
glupteba	0	0	0	0	0	0	0	0
hplocky	0	0	0	0	0	0	0	0
lethic	1	1	1	1	1	1	1	1
locky	202	193	156	183	29	29	19	19
midie	30	30	30	30	30	30	30	30
mikey	18	18	18	18	7	7	7	7
myxah	13	13	13	13	13	13	13	13
ngrbot	0	0	0	0	0	0	0	0
onion	1692	764	12	12	12	12	12	12

Continued on next page

Table 4.17 – continued from previous page

Family	hdbscan	hdbscan+rules	hdbscan+goodware	hdbscan+rules+ goodware	hdbscan:clot	hdbscan:clot+rules	hdbscan:clot+ goodware	hdbscan:clot+rules+ goodware
rack	0	0	0	0	0	0	0	0
razy	194	194	194	194	97	97	180	180
reconyc	0	0	0	0	0	0	0	0
ruskill	0	0	0	0	0	0	0	0
sage	0	0	0	0	0	0	0	0
sagecrypt	58	58	58	58	58	33	58	58
scar	36	36	36	36	36	36	36	36
scatter	35	35	1	35	35	35	1	35
score	0	0	0	0	0	0	0	0
shade	0	0	0	0	0	0	0	0
shiz	123	120	123	120	123	120	123	120
teerac	0	0	0	0	0	0	0	0
tescrypt	268	268	268	268	80	80	80	80
teslacrypt	1212	1212	1212	1212	1086	1086	1086	1086
tinba	9	9	9	9	9	9	9	9
torrentlocker	0	0	0	0	0	0	0	0
tpyn	1	1	1	1	1	1	1	1
upatre	31	34	23	34	32	34	32	31
waldek	0	0	0	0	0	0	0	0
yakes	141	139	141	139	141	140	141	140
zbot	10	10	10	10	10	10	10	10
zboter	0	0	0	0	0	0	0	0
zegost	0	0	0	0	0	0	0	0
zerber	435	435	367	435	435	435	325	435
zusy	156	153	156	153	124	147	124	147

Table 4.17: Table that shows the number of new malware positives found by the rules trained on the family on the row with the parameters indicated by the respective column. All the rules used for the tests have been generated using the Russell Rao distance.

Family	**hdbscan**	**hdbscan+rules**	**hdbscan+goodware**	**hdbscan+rules+ goodware**	**hdbscan:clot**	**hdbscan:clot+rules**	**hdbscan:clot+ goodware**	**hdbscan:clot+rules+ goodware**
agentb	1	1	1	1	1	1	1	1
allaple	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
aura	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
barys	3	3	3	3	3	3	3	3
beebone	1	1	1	1	1	1	1	1
bitman	714	675	714	675	714	675	714	675
carberp	0	0	0	0	0	0	0	0
cerber	240	235	240	235	240	235	240	235
cloud	0	0	0	0	0	0	0	0
coantor	0	0	0	0	0	0	0	0
critroni	0	0	0	0	0	0	0	0
crowti	292	256	91	91	532	285	86	86
crypmod	287	284	287	284	287	284	287	284
cryptowall	38	33	36	33	72	33	36	33
dagozill	4	4	4	4	1	1	1	1
dalexis	33	43	33	43	4	14	4	14
delf	0	0	0	0	0	0	0	0
deshacop	1	1	1	1	1	1	1	1
dmalocker	0	0	0	0	0	0	0	0
dridex	0	0	0	0	0	0	0	0
dynamer	5	5	5	5	5	5	5	5
enestaller	9	9	9	9	9	9	9	9
enestedel	4	4	4	4	4	4	4	4
fareit	2	1	2	1	2	1	2	1
fraudrop	0	0	0	0	0	0	0	0
gamarue	30	21	30	21	30	21	30	21
genericcryptor	1	1	1	1	1	1	1	1
genkryptik	4	4	4	4	4	4	4	4

Continued on next page

Table 4.18 – continued from previous page

Family	hdbscan	hdbscan+rules	hdbscan+goodware	hdbscan+rules+ goodware	hdbscan:clot	hdbscan:clot+rules	hdbscan:clot+ goodware	hdbscan:clot+rules+ goodware
glupteba	0	0	0	0	0	0	0	0
hplocky	0	0	0	0	0	0	0	0
lethic	1	1	1	1	1	1	1	1
locky	57	55	57	55	57	424	57	53
midie	24	22	24	22	24	22	24	22
mikey	218	187	218	187	143	112	143	112
myxah	13	13	13	13	13	13	13	13
ngrbot	0	0	0	0	0	0	0	0
onion	9	9	9	9	9	9	9	9
rack	0	0	0	0	0	0	0	0
razy	37	46	37	46	37	46	37	46
reconyc	0	0	0	0	0	0	0	0
ruskill	0	0	0	0	0	0	0	0
sage	0	0	0	0	0	0	0	0
sagecrypt	51	47	51	47	51	47	51	47
scar	35	35	35	35	35	35	35	35
scatter	6	7	6	7	7	7	7	7
score	0	0	0	0	0	0	0	0
shade	0	0	0	0	0	0	0	0
shiz	128	87	128	87	128	87	128	87
teerac	0	0	0	0	0	0	0	0
tescrypt	80	68	66	66	80	68	66	66
teslacrypt	759	701	759	701	758	702	758	702
tinba	9	9	9	9	9	9	9	9
torrentlocker	0	0	0	0	0	0	0	0
tpyn	1	10	1	10	1	10	1	10
upatre	54	72	39	20	54	72	39	20
waldek	2	2	2	2	0	2	0	2
yakes	133	121	133	121	133	121	133	121
zbot	6	6	6	6	6	6	6	6

Continued on next page

Table 4.18 – continued from previous page

Family	hdbscan	hdbscan+rules	hdbscan+goodware	hdbscan+rules+ goodware	hdbscan:clot	hdbscan:clot+rules	hdbscan:clot+ goodware	hdbscan:clot+rules+ goodware
zboter	0	0	0	0	0	0	0	0
zegost	0	0	0	0	0	0	0	0
zerber	312	300	312	300	312	300	312	300
zusy	167	135	167	135	170	135	170	135

Table 4.18: Table that shows the number of new malware positives found by the rules trained on the family on the row with the parameters indicated by the respective column. All the rules used for the tests have been generated using the Jaccard distance.

The results obtained by using the HDBSCAN algorithms are comparable with the ones in the previous section, with which they share the considerations made on the biggest families, however, for some small families, the number of new malware positives is slightly smaller. This is due to the clustering procedure, that tends to generate a high number of noise points, especially in small families, which lead ultimately to the creation of a rule per each, and, as a consequence, on over-specific rules.
It is worth mentioning that, for most of the families, the number of new malware positives is always similar when changing the parameters used for the training. There are, however, some notable exceptions, for which, when using the "clot" algorithm for the rule generation, the number of new positives decreases, probably due to more specific and balanced rules.
Finally, by looking at the results of the "onion" family in Table 4.17 and of the "crowti","locky" and "cryptowall" families in Table 4.18, it is evidenced a spike in the number of new positives found. This result, together with the number of false positives indicated respectively in Table 4.10 and in Table 4.11, underlines the importance of having specific rules (i.e., rules with a consistent number of features each).
Specifically, it is evidenced that, a spike in the number of new matches corresponds to a relatively high number of false positives. By manually checking the rule sets interested by this trend, it was noticed that each presents one or, at most, two rules with a relatively small number of literals (i.e., always below 100 features) and lacking references to relevant features(e.g., the sample's resources or, less relevant but still important, the rich signature's features). The lack of important features leads these rules to be composed

prevalently of imported functions, however, the packing mechanism tends to hide all the distinctive functions from the Import Address Table, leaving only the generic ones (i.e., the functions imported from the "kernel32" DLL), which are usually contained in the goodware's Import Address Table as well. Consequently, the rules will match a big number of malicious samples, but also a non-negligible number of benign ones.
To this extent, the addition of domain expert written rules as features is typically helpful to reduce the number of false positives, by adding targeted features, and consequently by reducing the number of new matches found. The reduction of the number of matches might be seen as a negative side of the introduction of new features, however, it is commonly preferred, in the IT security area, to have rules that tend to be slightly over-specific but with a low number of false positives than having generic rules that produce an high number of false positives.

4.5.3 yarGen true positives

Table 4.19 and Table 4.20 show the number of new true positives found by the rules generated by yarGen, applying all the possible parameters the tool disposes of. Table 4.20 differs from Table 4.19 in that all the trainings have been done by using the "z0" parameter. Taking a closer look to the results, and comparing them with the YaYaGenPE ones, it is evidenced an higher number of true positives, in particular when looking at the medium-small families (referring to Figure 4.1). Although this might be promising, it is necessary to confront these results with the false positives respectively in Table 4.12 and Table 4.13. Specifically, the high number of new matches, along with the high number of false positives, might be an indicator of too much generic rules, which are not tailored to detect a particular malware family and, typically, not usable in a real context due to the high number of false positives.

Family	**plain**	**goodware**	**excludegood**	**goodware+ excludegood**	**opcodes**	**opcodes+ goodware**	**opcodes+ excludegood**	**opcodes+ goodware+ excludegood**
agentb	12	6	12	6	12	6	12	6
allaple	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
aura	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
barys	1	1	1	1	1	1	1	1
beebone	2	2	2	2	1	1	1	1
bitman	877	931	877	931	332	345	332	345

Continued on next page

Table 4.19 – continued from previous page

Family	**plain**	**goodware**	**excludegood**	**goodware+ excludegood**	**opcodes**	**opcodes+ goodware**	**opcodes+ excludegood**	**opcodes+ goodware+ excludegood**
carberp	19	19	19	19	0	0	0	0
cerber	368	434	368	434	260	262	260	262
cloud	99	99	99	99	8	8	8	8
coantor	0	0	0	0	0	0	0	0
critroni	8	8	8	8	0	0	0	0
crowti	63	207	63	207	44	52	44	52
crypmod	409	371	409	371	75	90	75	90
cryptowall	88	190	88	190	10	33	10	33
dagozill	8	4	8	4	2	1	2	1
dalexis	36	36	36	36	8	8	8	8
delf	0	0	0	0	0	0	0	0
deshacop	0	0	0	0	0	0	0	0
dmalocker	0	0	0	0	0	0	0	0
dridex	0	0	0	0	0	0	0	0
dynamer	94	102	94	102	66	102	66	102
enestaller	65	65	65	65	27	27	27	27
enestedel	26	28	26	28	26	26	26	26
fareit	1	1	1	1	1	1	1	1
fraudrop	68	68	68	68	2	2	2	2
gamarue	35	54	35	54	5	6	5	6
genericcryptor	5	5	5	5	5	5	5	5
genkryptik	4	4	4	4	4	4	4	4
glupteba	66	66	66	66	38	38	38	38
hplocky	3	3	3	3	3	3	3	3
lethic	1	21	1	21	0	1	0	1
locky	292	409	292	409	93	69	93	69
midie	24	16	24	16	19	16	19	16
mikey	67	14	67	14	8	8	8	8
myxah	33	33	33	33	15	15	15	15
ngrbot	0	0	0	0	0	0	0	0
onion	39	39	39	39	34	34	34	34
rack	29	29	29	29	6	6	6	6
razy	343	390	343	390	64	60	64	60

Continued on next page

Table 4.19 – continued from previous page

Family	plain	goodware	excludegood	goodware+ excludegood	opcodes	opcodes+ goodware	opcodes+ excludegood	opcodes+ goodware+ excludegood
reconyc	0	0	0	0	0	0	0	0
ruskill	0	0	0	0	0	0	0	0
sage	13	13	13	13	13	13	13	13
sagecrypt	56	56	56	56	56	56	56	56
scar	34	34	34	34	16	16	16	16
scatter	95	333	95	333	74	333	74	333
score	2	2	2	2	2	2	2	2
shade	89	339	89	339	61	339	61	339
shiz	141	141	141	141	141	141	141	141
teerac	38	38	38	38	16	16	16	16
tescrypt	491	374	491	374	24	36	24	36
teslacrypt	603	603	603	603	487	487	487	487
tinba	3	9	3	9	3	9	3	9
torrentlocker	1	1	1	1	1	1	1	1
tpyn	1	1	1	1	1	1	1	1
upatre	62	110	62	110	26	31	26	31
waldek	0	2	0	2	0	0	0	0
yakes	116	116	116	116	114	114	114	114
zbot	0	3	0	3	0	1	0	1
zboter	0	0	0	0	0	0	0	0
zegost	0	0	0	0	0	0	0	0
zerber	378	548	378	548	178	406	178	406
zusy	208	313	208	313	59	74	59	74

Table 4.19: Table that shows the number of new positives obtained by the yarGen generated rules over the family indicated in each row and using the parameters in each column.

Family	plain	goodware	excludegood	goodware+ excludegood	opcodes	opcodes+ goodware	opcodes+ excludegood	opcodes+ goodware+ excludegood
agentb	13	13	13	13	13	13	13	13

Continued on next page

Table 4.20 – continued from previous page

Family	plain	goodware	excludegood	goodware+ excludegood	opcodes	opcodes+ goodware	opcodes+ excludegood	opcodes+ goodware+ excludegood
allaple	0	0	0	0	0	0	0	0
atraps	0	0	0	0	0	0	0	0
aura	0	0	0	0	0	0	0	0
autoit	0	0	0	0	0	0	0	0
barys	3	3	3	3	3	3	3	3
beebone	2	2	2	2	1	1	1	1
bitman	719	907	622	849	508	544	504	513
carberp	19	19	19	19	0	0	0	0
cerber	361	438	345	321	272	283	271	298
cloud	8	8	8	8	8	8	8	8
coantor	0	0	0	0	0	0	0	0
critroni	9	9	9	8	2	2	2	0
crowti	80	77	79	77	75	71	75	71
crypmod	194	208	194	205	86	90	86	87
cryptowall	83	126	32	33	39	41	30	31
dagozill	3	3	3	3	1	1	1	1
dalexis	36	36	36	36	8	8	8	8
delf	0	0	0	0	0	0	0	0
deshacop	0	0	0	0	0	0	0	0
dmalocker	0	0	0	0	0	0	0	0
dridex	0	0	0	0	0	0	0	0
dynamer	76	168	76	97	66	168	66	97
enestaller	65	65	65	65	27	27	27	27
enestedel	26	26	26	26	26	26	26	26
fareit	2	2	2	2	2	2	2	2
fraudrop	2	2	2	2	2	2	2	2
gamarue	78	208	68	93	45	104	44	44
genericcryptor	5	5	5	5	5	5	5	5
genkryptik	4	4	4	4	4	4	4	4
glupteba	40	40	40	40	38	38	38	38
hplocky	3	3	3	3	3	3	3	3
lethic	1	1	1	1	1	1	1	1
locky	112	476	55	422	71	248	53	242

Continued on next page

Table 4.20 – continued from previous page

Family	plain	goodware	excludegood	goodware+ excludegood	opcodes	opcodes+ goodware	opcodes+ excludegood	opcodes+ goodware+ excludegood
midie	24	24	24	24	24	24	24	24
mikey	8	8	8	8	8	8	8	8
myxah	21	21	21	21	15	15	15	15
ngrbot	0	0	0	0	0	0	0	0
onion	70	295	39	39	34	36	34	34
rack	29	29	29	29	6	6	6	6
razy	266	229	192	234	111	111	111	111
reconyc	0	0	0	0	0	1	0	0
ruskill	0	48	0	21	0	0	0	0
sage	13	13	13	13	13	13	13	13
sagecrypt	148	148	61	61	61	61	61	61
scar	21	21	21	21	17	19	17	19
scatter	99	161	77	99	74	167	74	98
score	2	2	2	2	2	2	2	2
shade	71	148	71	0	61	148	61	0
shiz	141	141	141	141	141	141	141	141
teerac	26	26	26	26	16	16	16	16
tescrypt	159	189	53	166	82	85	82	82
teslacrypt	912	912	774	774	693	693	683	683
tinba	9	161	9	9	9	9	9	9
torrentlocker	1	1	1	1	1	1	1	1
tpyn	1	1	1	1	1	1	1	1
upatre	82	126	33	62	36	37	30	32
waldek	0	0	0	0	0	0	0	0
yakes	163	163	140	140	137	137	137	137
zbot	11	11	9	11	11	11	9	11
zboter	0	0	0	0	0	0	0	0
zegost	0	0	0	0	0	0	0	0
zerber	379	326	361	313	276	266	269	219
zusy	131	200	107	123	121	121	120	121

Continued on next page

Table 4.20 – continued from previous page

Family	plain	goodware	excludegood	goodware+ excludegood	opcodes	opcodes+ goodware	opcodes+ excludegood	opcodes+ goodware+ excludegood

Table 4.20: Table that shows the number of new positives obtained by the yarGen generated rules over the family indicated in each row and using the parameters in each column. As opposed to Table 4.19, all the trainings have been performed by using the “z0” parameter together with the indicated ones.

4.5.4 yaBin true positives

yaBin, as opposed to yarGen, tends to produce, in most of the families, less true positive matches with respect to YaYaGenPE when using the “yara” parameter, which generates the most accurate ruleset among the two possible parameters it has. On the other hand, the “yaraHunt” parameter, which tries to generate rules that are generic enough to match other samples, has a much higher number of true positives found. However, as for the yarGen tool, this increment in the number of malwares found has to be weighed by the high number of false positives reported in Table 4.14, which indicates that those rules are not specific enough. Indeed, although these rules match lots of malicious samples, the number of false positives also suggest that they are too weak and imprecise to be effectively used in a real context.
The complete results of the malware positives found by yaBin are in Table 4.21.

Family	**yara**	**yaraHunt**
agentb	13	44
allaple	0	0
barys	3	3
beebone	1	4
bitman	528	2234
cerber	250	406
critroni	0	9
crowti	60	370
crypmod	46	2237
cryptowall	29	147
dagozill	0	12
dalexis	0	8
deshacop	0	1615

Continued on next page

Table 4.21 – continued from previous page

Family	**yara**	**yaraHunt**
dridex	0	14
dynamer	5	5
enestaller	9	27
enestedel	4	5
fareit	2	37
fraudrop	2	1827
gamarue	6	236
genkryptik	4	30
glupteba	0	0
lethic	1	86
locky	10	2187
midie	14	395
mikey	8	1877
onion	1	34
rack	0	8
razy	91	2024
reconyc	4	1396
sage	13	67
sagecrypt	61	320
scar	17	2488
scatter	7	37
shiz	141	442
teerac	0	0
tescrypt	43	2091
teslacrypt	694	1549
tinba	9	1446
tpyn	1	56
upatre	19	588
waldek	0	0
yakes	137	2046
zbot	7	22
zerber	234	411
zusy	91	2475

Table 4.21: Table that represents the number of new true positives found by the rules generated by yaBin with the parameters indicated in the respective column using, as training set, the family indicated in the respective row.

Considering both the true positives of the current section and the false positives introduced in section 4.4, yaBin results are better than yarGen ones and somehow comparable with YaYaGenPE results, at least for the rules generated when using the more restrictive parameter (i.e., “yara”). Specifically, while YaYaGenPE privileges the true positive matches, at the cost of some false positives found, the yaBin “yara” rules do not generate false positives, but present a much restricted number of new positives.

4.5.5 Evaluating precision of the rules

In order to get an idea of how precise the rules are, it has been computed the precision of the malware positives of the previous sections.
Rules precision is not a fundamental characteristic to evaluate, in that, what really matters in the context of malware detection is that the rules are properly capable of detecting malicious samples. Nevertheless, the rules precision gives an idea of how well do the rules capture the distinctive traits of each family they were trained on.
Before showing the results, it is needed to introduce how is the precision computed. In particular, as already mentioned, the dataset has been divided into families by using the most likely label provided by AVclass (i.e., the label with the highest score in the list returned by the tool). The training phase has then been done family by family. In order to compute the precision and, eventually, the recall, it would be ideal to perform a k-fold validation for each family of the training set, however, although accurate, given the dataset used in this work, the number of samples per family is, in many cases, too small to accurately perform a valuable k-fold validation, not to mention the required time to test all the tools in all the conditions. For this reason, the precision has been computed by considering, for each matching sample of the test set, the entire list of labels AVclass gave to it, and by checking whether that list contains the training family label or not.
These results, although not as accurate as the k-fold validation, give an idea of the accuracy of the tools when working with not perfectly clustered datasets. Specifically, due to the imprecise AVclass family attribution, two main classification errors may have been made: either the sample does not belong to the family it has been assigned to, due to the selection of a random AVclass label among those with the highest score, or, the family is an alias of another family. The former implies that, training the rules over a sample which is not really part of the attributed family might create rules that, actually, cover some samples of other families. The latter, instead, implies that samples are distributed among the two alias families and the families’ sizes strictly depend on the number of Anti-Virus that use one family label instead of the other. Nevertheless, in this case, rules trained on a specific family alias should be capable of matching a fair amount of samples belonging to the other alias.
Both these considerations can be evaluated by taking into account the complete list of family labels returned by AVclass: in the first case, indeed, if several samples are assigned to a particular family due to a split decision, it is likely that the samples of the other

family, if similar to the first ones, will present, in the list of AVclass labels, the name of the first family. In the second case, similarly, if a family is an alias of another, the majority of the samples of both families should have both the labels in their list of AVclass labels.
For these reasons, computing the precision basing on the complete list of AVclass labels is a valuable way to get an idea of how well do the rules work.
Nevertheless, these results are worsened by the fact that some rules, altough targeted for some specific malware families, will match similar samples of, possibly, families different from the training ones. For this reason *low precision values are not necessarily indicators of bad results* and, in order to get an idea of the real quality of the rules, it is introduced a relative comparison among the tools, which will evidence when, although presenting low precisions, results are still acceptable.
Table 4.22 shows the average precisions of the tools computed on the families containing at least 3 samples, by keeping constant the parameters indicated in the columns and averaging the single precision values determined by varying all the possible remaining parameters for all the tools. For a complete tabular representation of the precisions determined in all the training possibilities it is reminded to Appendix B.

	YaYaGenPE				**yarGen**		**yaBin**	
Family	**UDT Russell Rao**	**UDT Jaccard**	**HDB. Russell Rao**	**HDB. Jaccard**	**plain**	**z0**	**yara**	**yara Hunt**
agentb	0.0	0.0	0.0	0.0	4.2	7.7	7.7	2.3
allaple	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
barys	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
beebone	100.0	100.0	100.0	100.0	75.0	75.0	100.0	25.0
bitman	77.4	77.2	58.4	80.9	74.0	78.3	83.7	53.8
cerber	79.1	79.0	87.1	91.2	68.1	74.9	92.8	71.7
critroni	NaN	NaN	NaN	NaN	25.0	39.3	NaN	22.2
crowti	69.8	71.2	66.1	51.2	49.1	63.7	73.8	16.2
crypmod	20.9	18.4	16.4	6.7	11.4	14.1	26.1	2.4
cryptowall	22.2	21.9	30.9	16.3	30.7	29.0	27.6	24.5
dagozill	0.0	0.0	0.0	0.0	9.4	0.0	NaN	8.3
dalexis	50.0	50.0	66.7	38.2	25.0	25.0	NaN	50.0
deshacop	NaN	NaN	100.0	100.0	NaN	NaN	NaN	0.2
dridex	NaN	NaN	NaN	NaN	NaN	NaN	NaN	7.1
dynamer	0.0	0.0	0.0	0.0	1.2	0.8	0.0	0.0
enestaller	77.8	77.8	77.8	77.8	32.2	32.2	77.8	44.4
enestedel	6.0	6.0	50.0	50.0	18.9	19.2	50.0	40.0
fareit	50.0	56.2	50.0	75.0	100.0	50.0	50.0	2.7

Continued on next page

	YaYaGenPE				**yarGen**		**yaBin**	
Family	**UDT Russell Rao**	**UDT Jaccard**	**HDB. Russell Rao**	**HDB. Jaccard**	**plain**	**z0**	**yara**	**yara Hunt**
fraudrop	33.3	33.3	NaN	NaN	27.2	50.0	50.0	0.2
gamarue	21.5	32.0	17.6	12.1	16.4	9.1	50.0	3.4
genkryptik	25.0	25.0	25.0	25.0	25.0	25.0	25.0	6.7
glupteba	2.3	2.3	NaN	NaN	2.1	8.2	NaN	NaN
lethic	16.7	16.7	100.0	100.0	34.9	100.0	100.0	4.7
locky	22.3	16.0	33.2	23.8	12.0	13.6	72.7	1.6
midie	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.5
mikey	13.4	8.9	0.0	3.9	10.8	0.0	0.0	0.8
onion	14.7	14.7	6.3	0.0	2.8	3.1	0.0	5.9
rack	NaN	NaN	NaN	NaN	10.1	10.1	NaN	12.5
razy	80.5	81.2	82.1	59.1	47.7	40.4	39.6	21.8
reconyc	NaN	NaN	NaN	NaN	NaN	0.0	0.0	0.0
sage	15.4	15.4	NaN	NaN	15.4	15.4	15.4	29.9
sagecrypt	59.4	54.2	55.5	63.4	52.6	49.2	54.1	21.2
scar	100.0	100.0	47.2	45.7	73.5	87.8	100.0	0.7
scatter	0.0	0.0	0.0	0.0	1.9	2.5	0.0	0.0
shiz	91.4	91.6	81.9	83.9	90.8	90.8	90.8	66.3
teerac	NaN	NaN	NaN	NaN	0.0	0.0	NaN	NaN
tescrypt	45.6	46.0	41.2	44.8	40.2	40.7	39.5	34.4
teslacrypt	88.5	88.8	83.4	89.6	90.4	87.5	91.2	67.5
tinba	11.1	11.1	11.1	11.1	5.6	10.0	11.1	0.1
tpyn	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
upatre	31.8	33.9	19.6	22.4	23.7	23.2	26.3	3.4
waldek	NaN	NaN	NaN	50.0	0.0	NaN	NaN	NaN
yakes	51.8	48.6	42.4	43.0	47.0	46.0	48.2	3.6
zbot	60.0	60.0	27.3	71.4	66.7	16.1	14.3	40.9
zerber	61.6	62.1	61.1	68.8	54.7	70.0	74.8	63.3
zusy	32.7	33.0	20.5	19.3	14.2	19.8	27.5	3.8

Table 4.22: Table showing the average precision computed on the family indicated in each row by varying, for each tool configuration indicated in the corresponding column, all the possible parameters. Whenever an entry presents the value "NaN" it is meant that the tool was not capable of matching, for any parameter of the corresponding tool, any malicious sample in the dataset apart from the training samples.

Results in Table 4.22 show that YaYaGenPE tends, for most of the families, to perform

better than yarGen and worse than the yaBin “yara” configuration.
On the other hand, the precision of the rules generated by the yaBin “yaraHunt” configuration are usually much worse than the YaYaGenPE ones.

4.6 Training set coverage and number of samples per rule

This section covers two aspects that are not strictly related to the rule performances, but that are still relevant for the malware detection context: the capability of the tools to create rules that cover the entire training set, and the number of samples per rule. In particular, the former is an important aspect of any automatic signature generation tool since it is desirable, and even necessary, that the tool generates rules that are capable of covering all the samples the user provides. The latter is the inverse of the number of rules generated per training set. It is less relevant than the former, however, considering approximately the same number of true positive matches, it would be better to generate the least number of rules possible, i.e., to have the highest number of covered samples per rule.

4.6.1 Training set coverage

For readability reasons, it is here reported the number of uncovered samples per family, which is the exact opposite of the training set coverage, but gives an immediate idea of the performances of the tools.
Table 4.23 reports the maximum, average and minimum number of uncovered samples per family for each tool, considering all the possible parameters of the tools and, in case of YaYaGenPE, the two possible distance measures (i.e., Russell Rao and Jaccard distances). The coverage values have been computed considering only the families whose number of samples is at least three.
As Table 4.23 shows, YaYaGenPE provides an almost absolute coverage of all the families in the dataset, with the only exception of the “Zbot” family, which was not covered completely due to a tampered PE file that “pefile” was not capable of analyzing.
On the other hand, both yarGen and yaBin aren’t capable of completely covering entirely the families provided in the dataset and this behavior holds particularly for the yarGen tool, for which the number of uncovered samples reaches a maximum of 790 uncovered samples for the “Teslacrypt” family. Also the best case, which is usually associated with the usage of the “z0” parameter, reaches a maximum number of uncovered samples equal to 189, in case of the “Cerber” family.
yaBin supports a better coverage with respect to the yarGen tool, however, it also presents some families for which the complete coverage was not possible. Specifically, the “Locky” family presents a minimum number of uncovered samples equal to 10, while, in case of the “Cerber” family, the maximum number of uncovered samples rises up to 47.

Last considerations to be done for both yarGen and yaBin is the balance between the training set coverage and the number of true positives and false positives matched. In particular, for both yarGen and yaBin, it has been observed that the minimum number of uncovered samples is reached when using, respectively, the “z0” parameter and the “yaraHunt” one. Both these parameters, as already underlined in the previous sections, usually generate a set of rules that is typically unusable in a real context, due to the high number of false positives found. As a consequence, the number of uncovered samples associated to rules usable in a real context is respectively, the average one indicated in Table 4.23 for yarGen and the maximum one for yaBin.[1]

Family	**YaYaGenPE**			**yarGen**			**yaBin**		
	max	**avg**	**min**	**max**	**avg**	**min**	**max**	**avg**	**min**
agentb	0	0	0	1	0	0	0	0	0
allaple	0	0	0	0	0	0	0	0	0
barys	0	0	0	2	1	0	0	0	0
beebone	0	0	0	0	0	0	0	0	0
bitman	0	0	0	450	214	24	0	0	0
cerber	0	0	0	328	229	189	47	25	2
critroni	0	0	0	1	1	0	0	0	0
crowti	0	0	0	48	22	0	1	1	0
crypmod	0	0	0	7	2	0	0	0	0
cryptowall	0	0	0	13	4	0	0	0	0
dagozill	0	0	0	0	0	0	2	2	2
dalexis	0	0	0	1	0	0	1	1	1
deshacop	0	0	0	1	0	0	0	0	0
dridex	0	0	0	2	1	0	0	0	0
dynamer	0	0	0	0	0	0	1	1	0
enestaller	0	0	0	0	0	0	0	0	0
enestedel	0	0	0	0	0	0	0	0	0
fareit	0	0	0	1	0	0	0	0	0
fraudrop	0	0	0	0	0	0	0	0	0
gamarue	0	0	0	16	6	0	0	0	0
genkryptik	0	0	0	0	0	0	0	0	0
glupteba	0	0	0	0	0	0	1	0	0
lethic	0	0	0	22	8	0	0	0	0

Continued on next page

[1]yaBin presents two parameters only, so, the average number of uncovered samples is the average of just two values, the worse of which is the “yara” (un-)coverage, while the best is related to the “yaraHunt” parameter.

Family	**YaYaGenPE**			**yarGen**			**yaBin**		
	max	**avg**	**min**	**max**	**avg**	**min**	**max**	**avg**	**min**
locky	0	0	0	99	60	20	41	25	10
midie	0	0	0	1	0	0	0	0	0
mikey	0	0	0	1	0	0	0	0	0
onion	0	0	0	6	3	0	0	0	0
rack	0	0	0	0	0	0	0	0	0
razy	0	0	0	10	5	0	2	1	0
reconyc	0	0	0	0	0	0	0	0	0
sage	0	0	0	0	0	0	0	0	0
sagecrypt	0	0	0	3	2	0	0	0	0
scar	0	0	0	13	5	0	0	0	0
scatter	0	0	0	1	0	0	2	1	0
shiz	0	0	0	0	0	0	0	0	0
teerac	0	0	0	0	0	0	0	0	0
tescrypt	0	0	0	12	4	0	0	0	0
teslacrypt	0	0	0	790	404	22	0	0	0
tinba	0	0	0	1	0	0	0	0	0
tpyn	0	0	0	0	0	0	0	0	0
upatre	0	0	0	50	26	8	2	1	0
waldek	0	0	0	1	0	0	0	0	0
yakes	0	0	0	284	142	0	0	0	0
zbot	1	1	1	38	31	17	0	0	0
zerber	0	0	0	139	79	29	3	2	0
zusy	0	0	0	17	6	0	0	0	0

Table 4.23: Table showing the maximum,average and minimum number of uncovered samples for the rules generated on the family indicated in each row for the tool indicated in the corresponding column. Measures have been retrieved by considering, for all the tools, all the possible parameters they dispose of and, in case of YaYaGenPE, also both the supported distances (i.e., Russell Rao and Jaccard distances).

4.6.2 Number of covered samples per rule

This section analyzes the number of produced rules by each tool for the families containing at least three samples.
Specifically, as for the previous section, it is here reported the minimum, average and maximum amount of samples covered per rule for each tool, by testing all the possible parameters and distance measures possible.
The results are reported in Table 4.24.

Family	**YaYaGenPE**			**yarGen**			**yaBin**		
	min	**avg**	**max**	**min**	**avg**	**max**	**min**	**avg**	**max**
agentb	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
allaple	1.0	1.7	4.0	0.8	0.8	0.8	1.0	1.0	1.0
barys	1.0	1.0	1.0	1.0	1.3	2.0	1.0	1.0	1.0
beebone	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
bitman	3.0	4.2	6.2	0.9	1.2	1.6	2.1	2.1	2.1
cerber	3.9	5.7	9.0	1.6	2.1	3.1	3.4	3.4	3.4
critroni	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
crowti	1.8	2.2	2.7	1.0	1.4	2.0	1.8	1.8	1.8
crypmod	1.8	2.1	2.3	0.8	0.9	1.0	1.6	1.6	1.6
cryptowall	1.5	1.8	2.5	0.8	1.0	1.3	1.3	1.3	1.3
dagozill	1.3	1.5	2.0	1.0	1.0	1.0	2.0	2.0	2.0
dalexis	1.1	1.4	2.7	0.9	0.9	0.9	1.1	1.1	1.1
deshacop	1.3	1.8	2.7	0.8	0.8	0.8	1.1	1.1	1.1
dridex	1.0	1.2	1.5	0.8	0.9	1.0	1.0	1.0	1.0
dynamer	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
enestaller	1.0	1.5	3.0	0.8	0.9	1.0	3.0	3.0	3.0
enestedel	1.0	1.7	6.0	1.0	1.0	1.0	3.0	3.0	3.0
fareit	2.8	3.1	4.7	0.9	1.0	1.1	1.4	1.4	1.4
fraudrop	1.0	1.2	1.5	1.0	1.0	1.0	1.0	1.0	1.0
gamarue	1.7	2.1	2.8	0.8	0.9	1.0	1.2	1.2	1.2
genkryptik	1.0	1.3	3.0	0.8	0.9	1.0	1.5	1.5	1.5
glupteba	1.0	1.3	2.0	1.0	1.0	1.0	1.0	1.0	1.0
lethic	3.5	7.9	28.0	0.8	1.0	1.3	4.7	4.7	4.7
locky	2.6	2.8	3.1	1.0	1.1	1.2	1.2	1.2	1.2
midie	1.5	2.0	3.0	1.0	1.0	1.0	1.0	1.0	1.0
mikey	1.1	1.4	2.0	0.9	0.9	1.0	1.1	1.1	1.1
onion	2.0	2.1	2.5	0.9	1.0	1.0	1.0	1.0	1.0
rack	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
razy	1.7	2.0	2.5	0.9	1.1	1.2	1.5	1.5	1.5
reconyc	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
sage	1.0	1.3	2.0	0.8	0.8	1.0	2.0	2.0	2.0
sagecrypt	2.0	2.4	3.1	0.9	1.0	1.1	1.9	1.9	1.9
scar	2.5	3.1	4.2	0.8	1.0	1.6	2.1	2.1	2.1
scatter	1.2	1.8	4.0	0.9	1.0	1.0	1.0	1.0	1.0
shiz	2.1	2.3	2.6	0.8	0.9	1.0	1.8	1.8	1.8
teerac	1.0	1.2	1.5	1.0	1.0	1.0	1.0	1.0	1.0

Continued on next page

Family	YaYaGenPE			yarGen			yaBin		
	min	avg	max	min	avg	max	min	avg	max
tescrypt	1.7	2.0	2.4	0.9	0.9	1.0	1.4	1.4	1.4
teslacrypt	2.9	4.0	5.5	0.9	1.1	1.4	2.1	2.1	2.1
tinba	1.4	1.6	1.8	0.6	0.9	1.2	1.4	1.4	1.4
tpyn	1.6	2.0	2.6	0.9	0.9	0.9	1.6	1.6	1.6
upatre	1.6	2.0	2.5	0.9	1.0	1.0	1.2	1.2	1.2
waldek	1.0	1.2	1.8	1.0	1.0	1.2	1.0	1.0	1.0
yakes	3.0	3.4	4.0	1.2	1.4	1.6	2.8	2.8	2.8
zbot	2.7	3.1	3.6	1.9	2.1	2.3	2.0	2.0	2.0
zerber	3.8	5.5	10.3	1.3	1.4	1.7	3.7	3.7	3.7
zusy	1.9	2.2	2.6	0.8	0.9	1.0	1.5	1.5	1.5

Table 4.24: Table showing the maximum,average and minimum number of samples covered per rule on the family indicated in each row for the tool indicated in the corresponding column. Measures have been retrieved by considering, for all the tools, all the possible parameters they dispose of and, in case of YaYaGenPE, also both the supported distances (i.e., Russell Rao and Jaccard distances).

From the results it is evident how YaYaGenPE is the tool that has the highest samples per rule ratio, reaching a maximum of 28 samples per rule on the "Lethic" family. yarGen and yaBin, on the other hand, reach a maximum of, respectively, 3.1 on the "Cerber" family and 4.7 samples per rules on the "Lethic" family. On the same families, it is also possible to notice how YaYaGenPE covers, on average, a higher number of samples per rule with respect to the maximum number of the other two tools.
This also means that, in the majority of cases, YaYaGenPE generates a smaller number of rules per training set and, although it is not strictly fundamental, this feature is a measure of the scalability of each rule in terms of number of matched samples.

4.7 Final considerations on the signature generation tools

In order to get a better picture of the tools performances, it is necessary to consider all the previous sections together. Indeed, even tough true positives, false positives, precision and training set coverage have been analyzed separately, what really matters for an automatic signature generation tool is the combination of the four.
For the following considerations, it will be taken into account the overall behavior of yarGen, regardless of the "z0" parameter usage, and the yaBin "yara" rules for yaBin, since the "yaraHunt" parameter tends to generate results comparable and possibly even worse than the yarGen ones.

To this extent, by performing a general comparison of all the tools, it is possible to state that YaYaGenPE is the exact balance between yarGen and yaBin, in terms of true positives, false positives and overall precision. In particular, yarGen rules usually generate a high number of true positives as well as a non-negligible number of false positives and a precision lower than those of the other tools. On the other hand, the yaBin "yara" rules are much more accurate, in that, they find no false positives and present a fairly high precision, but they match a rather restricted number of true positives.
YaYaGenPE rules, as opposed to the other tools, present a small number of false positives, a high number of true positive matches and precision slightly lower than yaBin rules, but still comparable.
On the other hand, from the point of view of the training set coverage, YaYaGenPE seems to perform definitely better than both the other tools. On top of that, YaYaGenPE rules are also the ones with the highest samples per rule ratio.
Table 4.25 presents a final comparison of all the tools that have been extensively used during this work, with a view on both the performances and the different approaches used by each of them.

	YaraGenerator	**yarGen**	**yaBin**	**BASS**	**YaYaGenPE**
Pure YARA	YES	YES	YES	NO	YES
Based on	Strings	Strings	Binary	Binary	PE + rules
Algorithm	Common strings	Whitelist strings	Whitelist funcs	Bindiff + LCS	Set covering
Clustering	NO	NO	NO	YES	YES
Scalable	YES	YES	YES	NO	YES
Packing	NO	NO	YES	YES	YES
False positives	High	High	Low	Low	Low
High Coverage	NO	YES	YES	NO	YES
Training-set coverage	NO	NO	NO	YES	YES
Samples per rule	High	Low	Low	?	High

Table 4.25: Table presenting a complete comparison of all the tools analyzed during this work.

4.8 VirusTotal Intelligence platform tests

In this section there are a set of results obtained by testing some of the rules generated for the tests of the previous sections over the VirusTotal Intelligence platform.
VirusTotal [64] is, today, one of the biggest platforms for automatic and free software analysis. It also hosts the Intelligence platform that allows, among other things, to supply YARA rules which will be constantly applied to any new software uploaded to the platform in order to search for any match. Moreover, the Intelligence platform provides the RetroHunt service, which, given a set of YARA rules, tests them against their repository of binary data. The RetroHunt starts scanning the database backwards from the most recently provided software until either the rule set reaches a total number of 10,000 matches or a total of approximately 100 TeraBytes of data has been analyzed.
Results of the rules tested on the RetroHunt service are shown in Table 4.26.

Family	**Algorithm**	**# Positives**	**True positives**	**False Positives**
Olympic Destroyer	udt + greedy + rules	143	143	0
Sagecrypt	udt + clot + rules	136	136	0
Fareit	clot + rules	385	385	0
Scatter	udt + greedy	57	49	8
Scatter	udt + greedy + rules	35	31	4
Shiz	udt + clot + rules	12	12	0
Crowti	udt + greedy + rules	66	66	0
Tescrypt	udt (Jaccard) + clot + rules	1407	897	510

Table 4.26: Table representing the outcome of the RetroHunt procedure applied on the rules generated by the algorithm indicated in the respective column, over the family in the respective row. Wherever not specified differently, it is has been used the Russell Rao distance.

Most of the results shown are promising, in that, the number of matched positives is restricted, i.e., at most 1407 matches found on 100 TeraBytes, and the majority of them presents a high number of true positives and a low number of false positives. This is an

indicator of rules that are particularly targeted towards the family they have been trained on, since they are capable of filtering out most of the remaining software. Manual analysis on these results have also shown that, the majority of the true positives found presented, in the labels identified by AVclass, the corresponding training family, which is a strong hint of the accuracy of the rules.
Furthermore, focusing on the "Scatter" results, it is also possible to verify what was already mentioned when examining the number of False Positives found by the YaYaGenPE rules: when adding the user provided rules as features the number of False Positives tends to decrease.
Special considerations have to be done, nevertheless, for the "Tescrypt" family test, which reported false positives of approximately 36% of the total matches [2]. Although the number of false positives is rather low, considering the 100 TeraBytes analyzed on the platform, the ratio of true positives against false positives is not as good as expected. Further analysis have determined that the total number of false positives have been all found by a single rule, which, even if containing approximately 130 literals, did not present any of the features which are typically effective for malicious samples matching (i.e., resources related features and RICH header features). In particular, out of the 1407 positives, 1404 were due to the just mentioned rule, while the remaining 3 ones were matched by another one which targeted 3 true positives, specifically of the "Tescrypt" family.
Finally, for all the rules, performances similar to those in Table 4.26 were observerd on the on-line tests performed by the Intelligence platform on the daily provided software.

4.9 YaYaGenPE k-fold validation

In this section it is presented a set of k-fold validations performed over some of the biggest families in the dataset.
Before presenting the detailed results, it has to be mentioned that YaYaGenPE is *not* a data classifier, which means that the conventional k-fold validation measures do not hold in this case. For this reason, once divided each tested family into k partitions, as the k-fold validation requires, the training is performed on k-1 partitions and *only the recall* is computed over the remaining k-th partition, for each of the k tests required by the k-fold validation. Indeed, computing the precision on the k-th partition is, in this case, meaningless, since all the matched samples in the k-th partition are true positives of the tested family, i.e., the precision would be always equal to 1.
On the other hand, in order to give a meaningful idea of the precision, it is here reported the same precision illustrated in subsection 4.5.5, computed, for each of the k tests provided by the k-fold validation, on all the samples of the dataset minus those in the k-1 partitions

[2]Telling which of the matches are effectively False Positives is not trivial, so, it has been chosen to consider as False Positives any sample for which less than 3 Anti-Virus indicated it as malicious.

used for the training. In this case, the just mentioned precision takes into account both the samples of the k-th partition and the remaining dataset samples, which may still be of the tested family but assigned to another one or that may be of a completely different family.
High precision values thus mean that the tested family, on average, was capable of retrieving, among all the matched samples, lots of samples whose AVclass labels contain the tested family label, which should be an indicator of valuable rules.
Still, low precision values are *not necessarily* an indication of bad rules, because of two reasons: the rules are still matching true positives, even if of a different family, and, the rules have been trained to recognize common traits of the family, but *not* to distinguish that family from any other. In particular, for the latter, nothing prevents two families from presenting similar traits, at least from a static-analysis point of view.
Finally, as already mentioned several times, the family division performed by AVclass is not perfect and, as a consequence, the rules might have been trained on samples of the wrong family, which extends their coverage of the dataset to families different from the training one.
Results of the k-fold validations performed are illustrated in Table 4.27 and Table 4.28, respectively representing the results determined using the "Unsupervised Decision Tree" and the "HDBSCAN" clustering algorithms, in both cases using the "Russell Rao" distance only. Both the set of results were obtained by setting the value of k to 10.

Family	**Parameters**	**Recall**	**Overall precision**
yakes	greedy	0.699	0.675
yakes	greedy + rules	0.715	0.626
yakes	clot	0.704	0.683
yakes	clot + rules	0.721	0.655
teslacrypt	greedy	0.805	0.897
teslacrypt	greedy + rules	0.818	0.902
teslacrypt	clot	0.806	0.901
cerber	greedy	0.871	0.816
cerber	greedy + rules	0.888	0.814
cerber	clot	0.865	0.831
cerber	clot + rules	0.882	0.819
crowti	greedy	0.525	0.731
crowti	greedy + rules	0.538	0.731
crowti	clot	0.525	0.735
crowti	clot + rules	0.525	0.719
zusy	greedy	0.495	0.338
zusy	greedy + rules	0.51	0.328

Continued on next page

Family	Parameters	Recall	Overall precision
zusy	clot	0.436	0.41
zusy	clot + rules	0.495	0.373
upatre	greedy	0.358	0.428
upatre	greedy + rules	0.384	0.462
upatre	clot	0.325	0.521
upatre	clot + rules	0.339	0.519
zerber	greedy	0.879	0.724
zerber	greedy + rules	0.897	0.642
zerber	clot	0.882	0.733
zerber	clot + rules	0.885	0.668
shiz	greedy	0.491	0.878
shiz	greedy + rules	0.561	0.852
shiz	clot	0.471	0.872
shiz	clot + rules	0.55	0.864

Table 4.27: Table representing the recall and overall precision of the k-fold validation applied on the combination of family and parameters indicated in the first two columns. All the results have been determined by using the “Unsupervised Decision Tree” clustering algorithm.

Family	Parameters	Recall	Overall precision
yakes	greedy	0.73	0.612
yakes	greedy + rules	0.746	0.501
yakes	clot	0.723	0.66
yakes	clot + rules	0.743	0.511
cerber	greedy	0.888	0.76
cerber	greedy + rules	0.882	0.731
cerber	clot	0.882	0.763
cerber	clot + rules	0.884	0.752
crowti	greedy	0.539	0.585
crowti	greedy + rules	0.564	0.294
crowti	clot	0.539	0.545
crowti	clot + rules	0.564	0.199
zusy	greedy	0.567	0.199
zusy	greedy + rules	0.552	0.252
zusy	clot	0.507	0.213
zusy	clot + rules	0.552	0.253

Continued on next page

Family	Parameters	Recall	Overall precision
upatre	greedy	0.452	0.35
upatre	greedy + rules	0.466	0.361
upatre	clot	0.42	0.338
upatre	clot + rules	0.428	0.353
zerber	greedy	0.891	0.604
zerber	greedy + rules	0.9	0.589
zerber	clot	0.891	0.609
zerber	clot + rules	0.9	0.589
shiz	greedy	0.491	0.828
shiz	greedy + rules	0.519	0.804
shiz	clot	0.491	0.828
shiz	clot + rules	0.519	0.804

Table 4.28: Table representing the recall and overall precision of the k-fold validation applied on the combination of family and parameters indicated in the first two columns. All the results have been determined by using the “HDBSCAN” clustering algorithm.

Results show an opposite behavior of the “Unsupervised Decision Tree” rules with respect to the “HDBSCAN” ones. Specifically, the former presents higher overall precision values while the latter has better recall values. This pattern is somehow related to the clustering statistics in Table 4.2 and Table 4.3 in which, HDBSCAN presented slightly lower homogeneity values and slightly lower number of clusters generated. As a consequence, the rules generated by HDBSCAN are likely to be less specific than the “Unsupervised Decision Tree” ones, which means that they are capable of matching more samples than the last ones but that they are also slightly less precise.
Some general considerations can also be done by comparing Table 4.24 with the results in Table 4.27 and Table 4.28. Specifically, it is possible to see that the number of samples covered per rule is related to the recall of the family. In particular, families with high samples per rule are also likely to present high recall. This consideration is also, probably, linked to the similarity of the samples inside any family, i.e., families with similar samples usually require less rules to cover the entire dataset and, concurrently, the rules trained exclusively on part of the samples are likely to cover a consistent part of the remaining family samples. Vice versa, families consisting of samples really distant one from each other require lots of specific rules to cover the training set and, rules trained on part of the family samples, may not cover many other remaining samples of the same family.

Chapter 5

Conclusions

The aim of this thesis was the design and development of an automatic signature generation tool for the Microsoft Windows Operating System. This framework was created with the goal of aiding security experts by generating signatures that can capture common traits of the provided malicious samples.
Given the high ratio at which new malware are introduced, it is indeed evident that a manual analysis and signature generation for each new malware found is not even an option. For this reason, the developed framework has also the goal of being sufficiently scalable, allowing to handle thousands of samples in hours.
Scalability, indeed, is today one of the biggest concerns for the automatic signature generation, along with the creation of meaningful and targeted signatures. As a matter of fact, the state-of-the-art signature generation tools are not capable of dealing properly with both. As an example, the BASS framework has been proved to provide really accurate and specific signatures at the cost of an expensive signature generation procedure, due to an heavy clustering algorithm underneath. On the other hand, tools like YaraGenerator, yarGen and yaBin have linear complexity in the number of samples provided, but all of them have some flaws in the generation of targeted signatures. In this direction, YaYaGenPE aims at handling a consistent number of samples and simultaneously at producing accurate rules.
Results have shown that YaYaGenPE generated rules are, indeed, more accurate than the majority of the tested tools, while providing a better coverage of the provided samples and covering more samples per rule. Moreover, the rules tested on the VirusTotal Intelligence platform have shown promising results, making the tool suitable for usage in a real context, where the rules will have to face thousands of new samples daily.

5.1 Limitations and future work.

Although the design of the framework has shown already interesting performances, there is still room for improvement, specifically addressing the current tool limitations. In particular, there are still some points where the tool might handle things differently and probably even more accurately, which will be addressed in the next paragraphs.

Features selection. The current set of supported features, along with the binary encoding of each, can reach numbers in the order of tens of thousands features with datasets of thousands of samples. Even though, in all the tests performed, thousands of features have never been a big concern for the resulting clusters quality, the *curse of dimensionality* [1] might be troublesome when handling a huge set of samples, for which it might be interesting studying some feature selection procedures.
Many of the previous malware detection approaches reported in section 2.2 were performing dimensionality reduction either by selecting the most statistically relevant features or by performing some of the most common feature selection procedures (e.g., feature hashing or Principal-Component-Analysis).
Although the latter might be worth a try, the former, even if faster, could be somehow limited to the particular malware panorama when the statistics about the features have been extracted. This means that, at the high pace of the malware evolution, the statistics found in a previous point in time might not be relevant anymore in the next future, and this would require a continuous update of the best statistical features.
The features selection, apart from possibly improving the clustering quality, might as well improve the speed of the clustering procedure, and this might help extending the tool scalability.

Additional features support. Thanks to the independence of the YaYaGenPE clustering and rule generation procedures from the extracted features, the framework is easily extensible to support newer, and probably even better, set of features. For this reason, one of the next works might be the analysis of the efficacy of new malware features and their implementation in the tool.

Better goodwares management. As for the time being, goodwares are handled by creating more specific clusters from those whose signatures have generated some false positives. This approach is working in all the test cases that have been done throughout this thesis, however, it hinders the rules generality, since all the rules will be generated

[1]The *curse of dimensionality* is the phenomenon for which, when increasing the dataset dimensions, the distance between each point in the dataset tends to decrease. Extremely, for the number of dimensions that goes to infinity, the points distances tend to zero. This particular behavior affects particularly the Euclidean distances.

again even if only one of them was not specific enough.
For this reason, it is definitely worth studying a new, and more effective, mechanism of dealing with any false positive found during the rules generation procedure, in order to target the problematic rules only.

Rules quality evaluation and refinement. As shown in chapter 4, some of the rules contain prevalently too much generic literals and, consequently, tend to generate a non-negligible number of false positives. For this reason, the rules quality evaluation can be refined by assigning a proper weight to each of the features the tool extracts. By giving more importance to some features with respect to others, the rule generation might provide rules that focus on the features that are more relevant for the malware detection. As a side effect, this can possibly reduce the number of False Positives even without relying on a proper goodwares database.

Rules optimization. Most of the rules generated by YaYaGenPE contain a huge number of features each. This, although being good for avoiding false positives, might introduce over-specificity. For this reason, it is worth taking into account a final step of rules optimization that aims at keeping rules still accurate while reducing the number of features per rule. To this extent, YaYaGen [35] already disposes of a rule optimization algorithm that could be easily integrated in YaYaGenPE as well.

Appendix A

The PE File structure

The Portable Executable File Format is the standard executable format for all the Win32-based systems. This format is strongly based on the Unix's Common Object File Format (COFF) and takes into account some extensions needed to meet recent operating systems requirements (the COFF format was introduced in 1983 in UNIX System V) [41].
The documentation of the PE format is in the WINNT.H header file. In the following lines it will be explained the most important parts of the format accordingly to the purposes of this work.

A.1 The MS-DOS MZ format.

The Portable Executable always starts with the old MS-DOS MZ executable format. This format consists of a DOS Header, that describes the DOS executable, and of the DOS Stub, that is a set of instructions used to print an error message whenever the machine running the executable is not Win32-based.
The **DOS Header** is described in the IMAGE_DOS_HEADER structure of WINNT.H, and it is reported in Listing A.1 for commodity.

```
typedef struct _IMAGE_DOS_HEADER {
        WORD  e_magic;      /* 00: MZ Header signature */
        WORD  e_cblp;       /* 02: Bytes on last page of file */
        WORD  e_cp;         /* 04: Pages in file */
        WORD  e_crlc;       /* 06: Relocations */
        WORD  e_cparhdr;    /* 08: Size of header in paragraphs */
        WORD  e_minalloc;   /* 0a: Min. extra paragraphs needed */
        WORD  e_maxalloc;   /* 0c: Max. extra paragraphs needed */
        WORD  e_ss;         /* 0e: Initial (relative) SS value */
        WORD  e_sp;         /* 10: Initial SP value */
        WORD  e_csum;       /* 12: Checksum */
        WORD  e_ip;         /* 14: Initial IP value */
        WORD  e_cs;         /* 16: Initial (relative) CS value */
        WORD  e_lfarlc;     /* 18: RVA of relocation table */
        WORD  e_ovno;       /* 1a: Overlay number */
        WORD  e_res[4];     /* 1c: Reserved words */
        WORD  e_oemid;      /* 24: OEM identifier(for e_oeminfo) */
        WORD  e_oeminfo;    /* 26: OEM information; */
        WORD  e_res2[10];   /* 28: Reserved words */
        DWORD e_lfanew;     /* 3c: Offset to extended header */
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Listing A.1: DOS Header in WINNT.H

The first WORD[1] of the header is the so called magic number (e_magic, referring to the first field of Listing A.1), that represents the identification number of the DOS executable. This field contains the (hexadecimal) value 0x5A4D, which, interpreted as an array of characters, is the ASCII string 'MZ', as the initials of Mark Zbikowski, the creator of the first linker for DOS (please note that the array of characters is printed starting from the lowest index, that corresponds to the sequence {0x4D, 0x5A}). In the next fields there are important values to execute correctly the DOS executable (such as the initial stack segment value e_ss, or the initial instruction pointer value e_ip).
The most important field, however, is the e_lfanew DWORD, that represents the relative virtual address (RVA)[2] of the effective PE Header.
Immediately following the DOS Header there is the DOS Stub (starting at offset 0x40 from the beginning of the file). It is easy to take a look at the instructions executed there: by extracting the bytes from 0x40 up to the end of the DOS executable (typically, it occupies the first 128 bytes) and , by disassembling them (for example, by using ndisasm [3]), it is possible to see the operations performed. Basically, what the DOS Stub does is to print the string "This program cannot be run in DOS mode." and terminate.

The Rich Header. In between the DOS Header and the beginning of the PE Header, there usually is a gap that stores the so called "Rich Header". The name of this Header is due to the fact that, just before the end of the Header, there is always a DWORD containing the same 4 bytes whose ASCII sequence is "Rich", even though the preceding bytes may vary. This header is not officially documented, but it has been reverse engineered [42][25]. Results show that this header contains a list of items. Each element in the list

[1]In computer software programming, a WORD represents a 16 bits group. Other data definitions, in the same context, are: NIBBLE (to define a group of 4 bits), DWORD (or double WORD, a group of 32 bits) and QWORD (or quad word, a group of 64 bits).

[2]The relative virtual address is the offset, starting from the beginning of the file in memory, where the represented value can be found. The PE format strongly relies on RVAs, because they are easier to manage with respect to absolute addresses, especially in case of relocations

[3]ndisasm is one of the most widespread Linux x86 disassembler tools. It was originally designed to disassemble machine code produced by the NASM assembler, however, it works as well for any x86 executable code sequence.

consists of two DWORDs: the first identifies a compiling tool, while the second indicates how many times that tool has produced items for the executable.
The first DWORD can, in turn, be decomposed into 2 WORDs, the Most significant one containing the unique ID of the tool, the least significant one storing the build version of the tool [25]. An interesting fact about the Rich Header, as stated in [25] is that, during the 2018 Winter Olympics, a large scale malware attack used, among several other obfuscation techniques, a Rich Signature copied from malwares of the "Lazarus" group as an attempt to fool analysts into guessing the original authors.

A.2 The PE Header.

The PE header starts at the RVA indicated in the DWORD e_lfanew. At this address there is the "Signature" field, which is a DWORD that can assume different values, depending on the executable format. In case of a PE format, the field, interpreted as an array of characters, stands for the string "PE\0\0". Immediately after this field there is the File Header. This structure contains basic information about the file and it follows the original COFF implementations. Finally, the PE Header ends with the Optional Header, which, as opposed to what the name suggests, is mandatory, since it contains fundamental details of the executable.

A.2.1 File Header

The File Header is represented by an IMAGE_FILE_HEADER structure, whose representation is in Listing A.2.

```
typedef struct _IMAGE_FILE_HEADER {
        WORD  Machine;
        WORD  NumberOfSections;
        DWORD TimeDateStamp;
        DWORD PointerToSymbolTable;
        DWORD NumberOfSymbols;
        WORD  SizeOfOptionalHeader;
        WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Listing A.2: File Header structure.

Among the fields of the structure it is important to remark:

- the Machine WORD, that is a number identifying the CPU the file is destined to;
- the TimeDateStamp DWORD, that indicates when was the file generated; some AntiVirus might check this value in order to find suspicious instants of time, such as future ones.

- the NumberOfSections WORD, that indicates how many sections are in the file;
- the SizeOfOptionalHeader WORD, that is always fixed to 224 (bytes) and, as the name says, it tells the length of the Optional Header;
- the Characteristics WORD, that is used as a set of flags to give information about the executable. In particular, it is possible to specify whether the program is an executable (0x0002) or a DLL (0x2000) and many other values, not relevant for the purposes of this work.

A.2.2 Optional Header.

The Optional Header was initially introduced in the COFF format in order to include additional information when needed. However, the Portable Executable uses this Header to convey critical information. The header is represented by two different structures, depending on the processor architecture the file is destined to, respectively called IMAGE_OPTIONAL_HEADER32 and IMAGE_OPTIONAL_HEADER64. The two structures, apart from data type differences, have substantially the same set of fields, so, from here on, it will be described the 32 bits version. The complete representation of the header is in Listing A.3.

```
typedef struct _IMAGE_OPTIONAL_HEADER {

        /* Standard fields */

        WORD  Magic; /* 0x010b */
        BYTE  MajorLinkerVersion;
        BYTE  MinorLinkerVersion;
        DWORD SizeOfCode;
        DWORD SizeOfInitializedData;
        DWORD SizeOfUninitializedData;
        DWORD AddressOfEntryPoint;
        DWORD BaseOfCode;
        DWORD BaseOfData;
        DWORD ImageBase;
        DWORD SectionAlignment;
        DWORD FileAlignment;
        WORD  MajorOperatingSystemVersion;
        WORD  MinorOperatingSystemVersion;
        WORD  MajorImageVersion;
        WORD  MinorImageVersion;
        WORD  MajorSubsystemVersion;
        WORD  MinorSubsystemVersion;
        DWORD Win32VersionValue;
        DWORD SizeOfImage;
        DWORD SizeOfHeaders;
        DWORD CheckSum;
        WORD  Subsystem;
        WORD  DllCharacteristics;
        DWORD SizeOfStackReserve;
        DWORD SizeOfStackCommit;
        DWORD SizeOfHeapReserve;
        DWORD SizeOfHeapCommit;
        DWORD LoaderFlags;
        DWORD NumberOfRvaAndSizes;
        IMAGE_DATA_DIRECTORY
           DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Listing A.3: WINNT.H Optional Header representation (32 bits version)

A sufficiently detailed description of each field of the structure is in [41]. For this work it is enough to focus on few of those. In particular:

- SizeOfCode, SizeOfInitializedData, SizeOfUninitializedData fields, as the names explain, indicate the size of the respective sections (respectively, usually referred as .code or .text, .data and .bss). It is important that these sizes effectively match those of the sections, otherwise it might mean that the executable has been manipulated [57].

- The AddressOfEntryPoint field is the RVA of the first instruction to be executed by the program; usually, malwares or packers modify this address to make it point over an injected routine that, once it has performed its job (in case of packers, the decryption of the code), will jump back to the original entry point (OEP).

- The BaseOfCode and BaseOfData fields are the RVAs of, respectively, the code and data sections when mapped in memory.

- The ImageBase field represents the preferred load address of the file when it will be memory-mapped. Specifically, the linker, when creating an executable, assumes that the file will be loaded in memory to a predefined location. That location is indicated in this field, however, the loader might decide, for example because ASLR is enabled, to avoid loading the file in the specified address. In this last case, the loader has to perform the base relocations, using the .reloc section (more details following).

- The SectionAlignment field specifies that each section of the executable has to start at a virtual address that is multiple of that value, independently on the empty space that would appear in between two consecutive sections. As an example, let's assume that the alignment is set to be 0x1000 (which usually is, being it the standard size of a page); if the .code section occupies 256 bytes (0x0100 bytes) and starts at virtual address 0x1000, the next section has to begin at virtual address 0x2000 and the space in between will remain unused.

- The FileAlignment field has the same conceptual meaning of the previous field, but indicates the alignment when the file is stored on the secondary memory. For this reason, typically, this value is set to the size of a disk sector, that usually is 0x200 bytes (512 bytes).
- The SizeOfImage field indicates the total size of all the sections of the executable, aligned following the SectionAlignment value.
- The SizeOfHeaders field specifies the size of the DOS Header, the PE header and the section table, rounded up to the next multiple of FileAlignment value, that is, the entire size of the file minus the size of all the sections.
- The NumberOfRvaAndSizes field indicates the number of entries of the subsequent array of IMAGE_DATA_DIRECTORY structures. This entry has always been set to 16 since the introduction of the format.
- The DataDirectory element specifies an array IMAGE_DATA_DIRECTORY structures. Each of these structures consist of a VirtualAddress member, that specifies the RVA of the directory on the file, and of a Size member, that tells the dimension of the considered element (the structure is represented in Listing A.4). Some of these directory entries can be empty and some others, such as the Debug Directory, are superfluous and can be removed without causing any harm. However, the most important ones are the import directory and the Import Address Table (IAT). The former references to a structure that lists all the functions of external DLLs the executable relies on, while the latter is used by the program at runtime to find the addresses of these functions. It is up to the loader to set the addresses in the IAT before the program starts running[57].

```
typedef struct _IMAGE_DATA_DIRECTORY {
        DWORD VirtualAddress;
        DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Listing A.4: IMAGE_DATA_DIRECTORY structure

A.3 The Section Table.

Immediately following the PE Header, there is the section table. This table contains information about each section in the image. The sections here represented are sorted by starting RVAs. Each section consists of homogeneous data, and, depending on the content and purpose, has different access privileges.
The section table consists of an array of IMAGE_SECTION_HEADER (Listing A.5), whose number of entries is specified in the NumberOfSections field of the File Header.

```
typedef struct _IMAGE_SECTION_HEADER {
        BYTE  Name[IMAGE_SIZEOF_SHORT_NAME];
        union {
                DWORD PhysicalAddress;
                DWORD VirtualSize;
        } Misc;
        DWORD VirtualAddress;
        DWORD SizeOfRawData;
        DWORD PointerToRawData;
        DWORD PointerToRelocations;
        DWORD PointerToLinenumbers;
        WORD  NumberOfRelocations;
        WORD  NumberOfLinenumbers;
        DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

Listing A.5: IMAGE_SECTION_HEADER structure

The Section Header starts with an array of 8 bytes that specifies the Name of the section. Most of these names start with a dot (e.g., ".text" or ".code"), but it is not mandatory. However, it is necessary to remark that, in case the name takes all the 8 bytes, there will be no NULL terminator, since the string is interpreted as a list of bytes and not as a conventional one.
The following field is a union (Misc) whose meaning changes depending on the type of file that is being described (EXE or OBJs). If it is an EXE file, it will contain the size of the section (VirtualSize), before it is rounded up to the FileAlignment value. On the other hand, if it is an OBJ file, the union contains the PhysicalAddress of the section.
The VirtualAddress field contains, in an EXE file, the RVA where the loader should map the section. To get the absolute address where to map the section, it is necessary to take the ImageBase address (Optional Header field) and sum up this RVA. If the file is an OBJ, this field is meaningless and set to 0.
The SizeOfRawData field contains, for an EXE file, the size of the section after it has been rounded up to the next file alignment multiple. If it is set to 0, then it indicates uninitialized data.
The PointerToRawData is the file offset where the data of the section begins.
The Characteristics member is a set of flags that specify the section's attributes. The most important values of this field are:

- IMAGE_SCN_CNT_CODE (value 0x00000020), that means that the section contains code;
- IMAGE_SCN_CNT_INITIALIZED_DATA (value 0x00000040), that indicates that the section contains initialized data (this value is set for almost all the sections but the executable and the uninitialized ones);

- IMAGE_SCN_CNT_UNINITIALIZED_DATA (value 0x00000080), that indicates a section containing uninitialized data (typically named .bss);
- IMAGE_SCN_MEM_EXECUTE (value 0x20000000) specifies that the section can be executed (typically it is in conjunction with IMAGE_SCN_CNT_CODE);
- IMAGE_SCN_MEM_READ (value 0x40000000) indicates that the section is readable (typically all sections are);
- IMAGE_SCN_MEM_WRITE (value 0x80000000) specifies that the section is writable.

A.3.1 Most Common Sections.

In this section there will be a brief description of the sections that are usually present in almost every executable:

- **.text** or **.code**: this section is where, generally, all code emitted by the compiler or the assembler is stored. The linker, typically, takes all the **.text** sections, coming from the different OBJs of a project, and concatenates all of them into a single section. There might also be additional code inside this section, apart from the one written by the programmer. In particular, there might be instructions like:

  ```
  JMP DWORD PTR [XXXXXXXX].
  ```

  This happens when calling a function of an external module. In that case, the CALL instruction issued by the compiler does not transfer control directly to the external function, but it executes the previous instruction, which, in turn, jumps to an address whose value is stored in the .idata section, that is the real starting point of the function. This workaround has been introduced in order to reduce the number of locations where the loader has to patch the effective address of the DLL functions to be called. In particular, using pointers in the .idata section, only the values stored there have to be adjusted to point to the right function and not every single call to an external DLL function.

- **.data**: this section contains all the initialized data. This data comprehend global and static variables initialized at compile time. As for the .text section, the linker takes all the .data sections from the various OBJ and LIB files and combine them into a single section.
- **.bss**: this section is used to store all the uninitialized global or static variables. Again, one single .bss section is created by combining all the .bss sections in the OBJ and LIB files.

- **.idata**: this section contains information about the functions and data that the program imports from external DLLs. Each function imported by the PE is explicitly listed in this section.
- **.edata**: this section is a list of all the functions and data that the PE exports for other modules.
- **.reloc**: it is the section that holds a table of base relocations. A base relocation is an entry needed if the loader was not able to load the file where the linker assumed it would. In this case only, the .reloc section is used to find where, in the image, there is a 32-bit absolute address to be fixed in order to point to the right position.

A.4 An in-depth view of the Import procedure.

The ".idata" section contains all the information relative to the functions of external modules needed by an executable. It is here where the loader finds what is needed in order to patch the JMP instruction mentioned in the previous section.
The .idata section always starts with an array of IMAGE_IMPORT_DESCRIPTOR structures, whose fields are described in Listing A.6.

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
        union {
                DWORD   Characteristics;
                DWORD   OriginalFirstThunk;
        } DUMMYUNIONNAME;
        DWORD   TimeDateStamp;
        DWORD   ForwarderChain; /* -1 if no forwarders */
        DWORD   Name;
        /* RVA to IAT (if bound this IAT has actual addresses) */
        DWORD   FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR,*PIMAGE_IMPORT_DESCRIPTOR;
```

Listing A.6: IMAGE_IMPORT_DESCRIPTOR structure

This array does not have an explicitly specified length and, in order to specify the end, there is an element with all the fields set to NULL.
The Name field indicates which is the DLL the descriptor refers to.
The first union in Listing A.6 was originally devoted to indicate a set of flags. Since the PE File Format introduction, however, it has been used as the RVA to the first IMAGE_THUNK_DATA structure of an array terminated by a zero value element.
The last field, FirstThunk, can be interpreted in the same way as the first one. In summary, what happens is that the import descriptor will have two parallel arrays, zero terminated, composed of IMAGE_THUNK_DATA structures. The structure of an IMAGE_THUNK_DATA is represented in Listing A.7.

```
    typedef struct _IMAGE_THUNK_DATA32 {
            union {
                    DWORD ForwarderString;
                    DWORD Function;
                    DWORD Ordinal;
                    DWORD AddressOfData;
            } u1;
    } IMAGE_THUNK_DATA32,*PIMAGE_THUNK_DATA32;
```

Listing A.7: IMAGE_THUNK_DATA structure

Basically, its union ends up being filled either with the Ordinal of the imported API, or with the pointer to an IMAGE_IMPORT_BY_NAME structure. This last structure contains, in turn, a string naming the function to import and an hint on the ordinal of the function in the Export Table of the DLL (Listing A.8).

```
    typedef struct _IMAGE_IMPORT_BY_NAME {
            WORD    Hint;
            BYTE    Name[1]; /** "1" is just a placeholder. **/
    } IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

Listing A.8: IMAGE_IMPORT_BY_NAME structure

The reason for two arrays of the same type is that the first one (OriginalFirstThunk) is never modified, and, for this reason is usually called hint-name table. The second one (FirstThunk), on the contrary, is overwritten by the PE loader, in a way that, after the function has been found on the Export Table, the IMAGE_THUNK_DATA's union will contain the corresponding address. Getting back to the code section, the address in each "JUMP DWORD PTR [XXXXXXXX]" instruction is exactly the RVA of an IMAGE_THUNK_DATA in the FirstThunk array, that, once modified by the loader, will contain the right function pointer.
A graphical representation of an import descriptor and the related structures is in Figure A.1.

A.5 The Export Table.

As explained in the previous section, the PE loader accesses the Export Table of the requested module in order to patch the jump address inside the FirstThunk union. This table is usually stored in a section named ".edata".
The just mentioned section always begins with an IMAGE_EXPORT_DIRECTORY structure, whose fields are in Listing A.9. The structure's virtual address is then addressed by the IMAGE_DATA_DIRECTORY relative to the export table inside the DataDirectory array immediately following the Optional Header (Listing A.4).

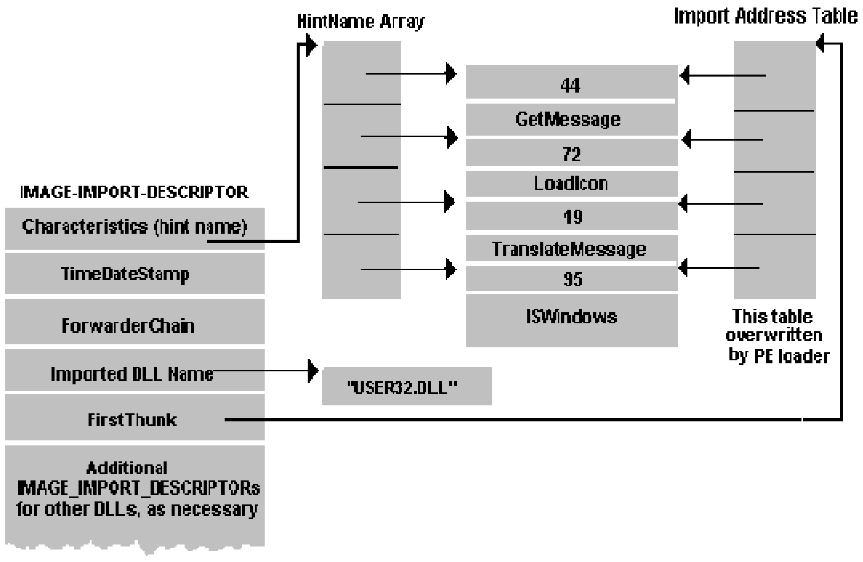


Figure A.1: Graphical representation of a .idata section entry.

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
        DWORD   Characteristics; /* unused */
        DWORD   TimeDateStamp;  /* time/date stamp indicating
                                   when the file was created*/
        WORD    MajorVersion;   /* unused */
        WORD    MinorVersion;   /* unused */
        DWORD   Name;   /* RVA to string Name of this DLL. */
        DWORD   Base;   /* Stores the starting ordinal number */
        DWORD   NumberOfFunctions;
        DWORD   NumberOfNames;
        DWORD   AddressOfFunctions;
        DWORD   AddressOfNames;
        DWORD   AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY,*PIMAGE_EXPORT_DIRECTORY;
```

Listing A.9: IMAGE_EXPORT_DIRECTORY structure.

It is important to remark the last three DWORDs, since they represent RVAs to three parallel arrays containing all the informations needed for each function inside the DLL. Each of these arrays has a number of elements corresponding to the value indicated in the NumberOfFunctions field (actually, the AddressOfNames array has "NumberOfNames" entries, but NumberOfNames and NumberOfFunctions seem to be always identical). Each array conveys a different type of information:

- the array pointed by AddressOfFunctions contains a sequence of function entry points;
- the array pointed by AddressOfNames has a sequence of pointers to ASCII strings,

each indicating the name of the function;

- the array pointed by AddressOfNameOrdinals stores an ordinal number for each function exported by the DLL.

A graphical representation of the Export Table is in Figure A.2.

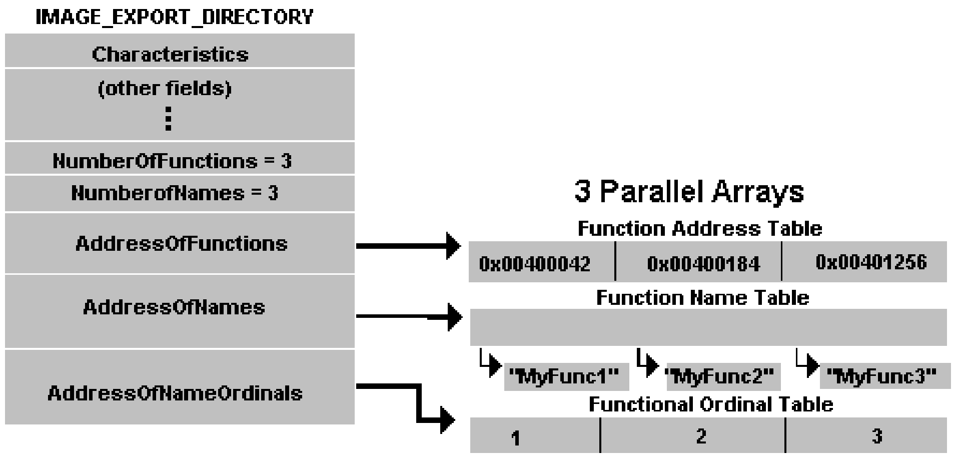


Figure A.2: Graphical representation of the .edata section.

A.6 The resources directory.

Typically, the resources are stored in a section named ".rsrc", even though this is not mandatory. This section, or whatever section is in charge of storing resources, contains a structure organized in a directory tree.
The root of this directory tree is addressed by the VirtualAddress field of one of the IMAGE_DATA_DIRECTORY entries in the DataDirectory array of the Optional Header (refer to Listing A.3). Each directory in the tree is represented by an IMAGE_RESOURCE _DIRECTORY structure, whose format is in Listing A.10.

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
        DWORD   Characteristics;
        DWORD   TimeDateStamp;
        WORD    MajorVersion;
        WORD    MinorVersion;
        WORD    NumberOfNamedEntries;
        WORD    NumberOfIdEntries;
        /*  IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[]; */
} IMAGE_RESOURCE_DIRECTORY,*PIMAGE_RESOURCE_DIRECTORY;
```

Listing A.10: IMAGE_RESOURCE_DIRECTORY structure.

Immediately following the directory structure there is an array of directory entries (IMAGE_RESOURCE_DIRECTORY_ENTRY structures), whose size is the sum of the NumberOfNamedEntries and of the NumberOfIdEntries fields. In particular, the entries identified by a Name appear first in the array, immediately followed by those identified by Id (same order as the fields in the resource directory structure). Each IMAGE_RESOURCE_DIRECTORY_ENTRY element has a structure described in Listing A.11 (for readability reasons it has been omitted the detailed description, that considers the Big-Endian equivalent of some fields).

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
        union {
                struct {
                        unsigned NameOffset:31;
                        unsigned NameIsString:1;
                } DUMMYSTRUCTNAME;
                DWORD   Name;
                WORD    Id;
                WORD    __pad;
        } DUMMYUNIONNAME;
        union {
                DWORD   OffsetToData;
                struct {
                        unsigned OffsetToDirectory:31;
                        unsigned DataIsDirectory:1;
                } DUMMYSTRUCTNAME2;
        } DUMMYUNIONNAME2;
} IMAGE_RESOURCE_DIRECTORY_ENTRY,*PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

Listing A.11: IMAGE_RESOURCE_DIRECTORY_ENTRY structure.

In short, the struct consists of two unions, which basically end up being a DWORD each, whose meaning depends on the most significant bit. The first one either contains a pointer to the string identifying the name of the entry or the Id of the entry (if the most significant bit is set to 1 then the other 31 bits are an offset, inside the file, of where the name starts). The second union is used to determine whether the entry is a leaf node or a sub-directory: if the most significant bit of that union is set to 1, then the entry represents a sub-directory and the remaining 31 bits point to another IMAGE_RESOURCE_DIRECTORY structure, otherwise the union points to an IMAGE_RESOURCE_DATA_ENTRY structure, that is the object containing details about the resource (Listing A.12).

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {
        DWORD   OffsetToData;
        DWORD   Size;
        DWORD   CodePage;
        DWORD   Reserved;
} IMAGE_RESOURCE_DATA_ENTRY,*PIMAGE_RESOURCE_DATA_ENTRY;
```

Listing A.12: IMAGE_RESOURCE_DATA_ENTRY structure.

The only important fields of that structure, according to the purposes of this work are the OffsetToData and the Size ones.

Directory tree layout. Given the easily growing complexity of the tree, the tree layout has been usually organized to follow a three level structure[66]. Specifically, the levels convey the following information:

- the first level of sub-directories, starting from the root, indicates the **type** of the resources stored inside: there is one directory entry per type of resource;
- the second level of sub-directories (the ones inside the "type" directories) contains a directory entry per resource **name**: there can be several resources having the same name, and all of them are stored inside the same directory;
- finally, at the third level, there is an IMAGE_RESOURCE_DIRECTORY_ENTRY for each **language** the resource is encoded into. In particular, it might happen that some resources can support different languages, even though they are conceptually equivalent. In this case, the Name field of the struct is interpreted as 2 numbers:
  - the **language identifier**, consisting of the lowest 10 bits of the Name DWORD;
  - the **sub-language identifier**, described by the remaining 22 bits.

Appendix B

Exhaustive precision results

This chapter presents all the precisions computed for all the tests performed on each family and analyzed tool. In the following tables there will not be the tables associated to the yaBin tool since it consits only of two parameters that generate YARA rules and, consequently, the precisions represented in Table 4.22 are already the effective results (i.e., not an average value). Whenever indicated the value "NaN" it is meant that the rules did not found any other positive apart from the training set ones. This happens, at least for YaYaGenPE, specifically on most of the smallest families (i.e., those with less than 10 elements), for which rules are really specific.
Finally, it has to be reminded that the precision here reported is intended as described in subsection 4.5.5.

B.1 YaYaGenPE precision results

B.1.1 Unsupervised Decision Tree Precision tables.

This section presents the results obtained by the "UDT" clustering algorithm on top of the rules generation procedures.

Family	**Size**	**udt**	**udt+rules**	**udt+goodware**	**udt+rules+ goodware**	**udt:clot**	**udt:clot+rules**	**udt:clot+ goodware**	**udt:clot+rules+ goodware**
agentb	3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
allaple	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
atraps	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Continued on next page

Table B.1 – continued from previous page

Family	Size	udt	udt+rules	udt+goodware	udt+rules+ goodware	udt:clot	udt:clot+rules	udt:clot+ goodware	udt:clot+rules+ goodware
aura	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
autoit	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
barys	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
beebone	3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
bitman	1090	0.791	0.744	0.789	0.774	0.797	0.781	0.735	0.783
bunitu	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
carberp	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
cerber	533	0.829	0.78	0.747	0.76	0.843	0.767	0.84	0.762
cloud	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
coantor	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
critroni	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
crowti	75	0.712	0.689	0.726	0.677	0.677	0.689	0.707	0.707
crypmod	42	0.163	0.165	0.155	0.367	0.167	0.367	0.125	0.165
cryptowall	47	0.276	0.16	0.276	0.16	0.276	0.16	0.276	0.192
dagozill	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
dalexis	8	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
delf	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
deshacop	8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dmalocker	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dridex	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dynamer	3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
enestaller	3	0.778	0.778	0.778	0.778	0.778	0.778	0.778	0.778
enestedel	6	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06
fareit	14	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
fraudrop	3	0.333	0.333	NaN	0.333	0.333	0.333	0.333	0.333
gamarue	33	0.182	0.4	0.182	0.2	0.133	0.167	0.167	0.286
genericcryptor	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
genkryptik	3	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
glupteba	4	0.023	0.023	0.023	0.023	0.023	0.023	0.023	0.023
hplocky	2	0.667	0.667	0.667	0.667	0.667	0.667	0.667	0.667
lethic	28	0.167	0.167	0.167	0.167	0.167	0.167	0.167	0.167
locky	647	0.429	0.077	0.147	0.378	0.149	0.116	0.126	0.361

Continued on next page

Table B.1 – continued from previous page

Family	Size	udt	udt+rules	udt+goodware	udt+rules+ goodware	udt:clot	udt:clot+rules	udt:clot+ goodware	udt:clot+rules+ goodware
midie	6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
mikey	8	0.0	0.357	0.0	0.0	0.0	0.357	0.0	0.357
myxah	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ngrbot	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
onion	40	0.214	0.083	0.214	0.077	0.214	0.083	0.214	0.077
rack	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
razy	40	0.808	0.8	0.808	0.809	0.808	0.816	0.79	0.8
reconyc	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ruskill	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
sage	4	0.154	0.154	0.154	0.154	0.154	0.154	0.154	0.154
sagecrypt	47	0.655	0.488	0.667	0.488	0.655	0.645	0.667	0.488
scar	25	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
scatter	12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
score	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
shade	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
shiz	104	0.914	0.903	0.909	0.912	0.922	0.922	0.917	0.91
teerac	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
tescrypt	29	0.45	0.481	0.45	0.45	0.45	0.45	0.465	0.45
teslacrypt	2478	0.892	0.893	0.893	0.887	0.865	0.885	0.878	0.893
tinba	7	0.111	0.111	0.111	0.111	0.111	0.111	0.111	0.111
torrentlocker	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
tpyn	13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
upatre	154	0.381	0.265	0.407	0.273	0.231	0.357	0.32	0.31
waldek	7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
yakes	682	0.524	0.486	0.526	0.525	0.526	0.546	0.526	0.486
zbot	43	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6
zboter	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
zegost	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
zerber	329	0.629	0.612	0.637	0.602	0.606	0.602	0.639	0.602
zusy	65	0.365	0.309	0.295	0.309	0.295	0.377	0.354	0.309

Continued on next page

Table B.1 – continued from previous page

Family	Size	udt	udt+rules	udt+goodware	udt+rules+goodware	udt:clot	udt:clot+rules	udt:clot+goodware	udt:clot+rules+goodware

Table B.1: Table that shows the precision of the rules trained on the family on the row with the parameters indicated by the respective column. All the rules used for the tests have been generated using the Russell Rao distance.

Family	Size	udt	udt+rules	udt+goodware	udt+rules+goodware	udt:clot	udt:clot+rules	udt:clot+goodware	udt:clot+rules+goodware
agentb	3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
allaple	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
atraps	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
aura	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
autoit	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
barys	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
beebone	3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
bitman	1090	0.785	0.758	0.748	0.775	0.784	0.764	0.758	0.807
bunitu	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
carberp	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
cerber	533	0.769	0.818	0.76	0.783	0.789	0.822	0.798	0.781
cloud	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
coantor	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
critroni	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
crowti	75	0.742	0.707	0.726	0.677	0.707	0.707	0.719	0.707
crypmod	42	0.125	0.165	0.167	0.165	0.157	0.165	0.16	0.367
cryptowall	47	0.276	0.192	0.25	0.16	0.276	0.16	0.276	0.16
dagozill	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
dalexis	8	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
delf	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Continued on next page

Table B.2 – continued from previous page

Family	Size	udt	udt+rules	udt+goodware	udt+rules+ goodware	udt:clot	udt:clot+rules	udt:clot+ goodware	udt:clot+rules+ goodware
deshacop	8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dmalocker	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dridex	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dynamer	3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
enestaller	3	0.778	0.778	0.778	0.778	0.778	0.778	0.778	0.778
enestedel	6	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.06
fareit	14	0.5	0.5	0.5	0.5	0.5	0.5	0.5	1.0
fraudrop	3	0.333	0.333	0.333	0.333	0.333	0.333	NaN	NaN
gamarue	33	0.167	0.5	0.333	0.25	0.167	0.5	0.143	0.5
genericcryptor	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
genkryptik	3	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
glupteba	4	0.023	0.023	0.023	0.023	0.023	0.023	0.023	0.023
hplocky	2	0.667	0.667	0.667	0.667	0.667	0.667	0.667	0.667
lethic	28	0.167	0.167	0.167	0.167	0.167	0.167	0.167	0.167
locky	647	0.053	0.118	0.118	0.343	0.046	0.357	0.132	0.112
midie	6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
mikey	8	0.0	0.0	0.0	0.357	0.0	0.357	0.0	0.0
myxah	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ngrbot	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
onion	40	0.214	0.083	0.214	0.077	0.214	0.083	0.214	0.077
rack	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
razy	40	0.812	0.819	0.816	0.817	0.816	0.807	0.812	0.8
reconyc	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ruskill	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
sage	4	0.154	0.154	0.154	0.154	0.154	0.154	0.154	0.154
sagecrypt	47	0.571	0.488	0.571	0.488	0.667	0.488	0.571	0.488
scar	25	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
scatter	12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
score	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
shade	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
shiz	104	0.924	0.91	0.922	0.91	0.909	0.91	0.931	0.91
teerac	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Continued on next page

Table B.2 – continued from previous page

Family	**Size**	**udt**	**udt+rules**	**udt+goodware**	**udt+rules+ goodware**	**udt:clot**	**udt:clot+rules**	**udt:clot+ goodware**	**udt:clot+rules+ goodware**
tescrypt	29	0.45	0.45	0.465	0.45	0.471	0.471	0.471	0.45
teslacrypt	2478	0.896	0.891	0.895	0.895	0.882	0.868	0.896	0.882
tinba	7	0.111	0.111	0.111	0.111	0.111	0.111	0.111	0.111
torrentlocker	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
tpyn	13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
upatre	154	0.375	0.273	0.478	0.385	0.318	0.265	0.304	0.312
waldek	7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
yakes	682	0.509	0.515	0.505	0.459	0.51	0.48	0.466	0.447
zbot	43	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6
zboter	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
zegost	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
zerber	329	0.609	0.601	0.615	0.614	0.621	0.646	0.597	0.664
zusy	65	0.319	0.33	0.354	0.309	0.365	0.309	0.354	0.301

Table B.2: Table that shows the precision of the rules trained on the family on the row with the parameters indicated by the respective column. All the rules used for the tests have been generated using the Jaccard distance.

B.1.2 HDBSCAN Precision tables.

This section presents the precision values obtained applying the "HDBSCAN" clustering algorithm on top of the rules generation procedures.

Family	**Size**	**hdbscan**	**hdbscan+rules**	**hdbscan+goodware**	**hdbscan+rules+ goodware**	**hdbscan:clot**	**hdbscan:clot+rules**	**hdbscan:clot+ goodware**	**hdbscan:clot+rules+ goodware**
agentb	3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Continued on next page

Table B.3 – continued from previous page

Family	**Size**	**hdbscan**	**hdbscan+rules**	**hdbscan+goodware**	**hdbscan+rules+ goodware**	**hdbscan:clot**	**hdbscan:clot+rules**	**hdbscan:clot+ goodware**	**hdbscan:clot+rules+ goodware**
allaple	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
atraps	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
aura	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
autoit	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
barys	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
beebone	3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
bitman	1090	0.585	0.584	0.585	0.584	0.585	0.584	0.585	0.584
carberp	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
cerber	533	0.828	0.84	0.828	0.84	0.9	0.915	0.9	0.915
cloud	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
coantor	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
critroni	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
crowti	75	0.608	0.608	0.712	0.608	0.689	0.689	0.689	0.689
crypmod	42	0.164	0.164	0.164	0.164	0.164	0.164	0.164	0.164
cryptowall	47	0.317	0.3	0.317	0.3	0.317	0.3	0.317	0.3
dagozill	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
dalexis	8	0.667	0.667	0.667	0.667	0.667	0.667	0.667	0.667
delf	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
deshacop	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
dmalocker	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dridex	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dynamer	3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
enestaller	3	0.778	0.778	0.778	0.778	0.778	0.778	0.778	0.778
enestedel	6	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
fareit	14	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
fraudrop	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
gamarue	33	0.167	0.25	0.167	0.167	0.167	0.167	0.158	0.167
genericcryptor	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
genkryptik	3	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
glupteba	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Continued on next page

Table B.3 – continued from previous page

Family	**Size**	**hdbscan**	**hdbscan+rules**	**hdbscan+goodware**	**hdbscan+rules+ goodware**	**hdbscan:clot**	**hdbscan:clot+rules**	**hdbscan:clot+ goodware**	**hdbscan:clot+rules+ goodware**
hplocky	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
lethic	28	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
locky	647	0.089	0.093	0.109	0.098	0.448	0.448	0.684	0.684
midie	6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
mikey	8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
myxah	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ngrbot	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
onion	40	0.001	0.001	0.083	0.083	0.083	0.083	0.083	0.083
rack	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
razy	40	0.814	0.814	0.814	0.814	0.804	0.804	0.85	0.85
reconyc	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ruskill	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
sage	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
sagecrypt	47	0.552	0.534	0.534	0.534	0.552	0.667	0.534	0.534
scar	25	0.472	0.472	0.472	0.472	0.472	0.472	0.472	0.472
scatter	12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
score	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
shade	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
shiz	104	0.813	0.825	0.813	0.825	0.813	0.825	0.813	0.825
teerac	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
tescrypt	29	0.362	0.362	0.362	0.362	0.462	0.462	0.462	0.462
teslacrypt	2478	0.815	0.815	0.815	0.815	0.853	0.853	0.853	0.853
tinba	7	0.111	0.111	0.111	0.111	0.111	0.111	0.111	0.111
torrentlocker	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
tpyn	13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
upatre	154	0.194	0.176	0.217	0.176	0.219	0.176	0.219	0.194
waldek	7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
yakes	682	0.418	0.424	0.418	0.424	0.426	0.429	0.426	0.429
zbot	43	0.273	0.273	0.273	0.273	0.273	0.273	0.273	0.273
zboter	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Continued on next page

Table B.3 – continued from previous page

Family	Size	hdbscan	hdbscan+rules	hdbscan+goodware	hdbscan+rules+ goodware	hdbscan:clot	hdbscan:clot+rules	hdbscan:clot+ goodware	hdbscan:clot+rules+ goodware
zegost	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
zerber	329	0.584	0.584	0.651	0.584	0.584	0.584	0.735	0.584
zusy	65	0.192	0.196	0.192	0.196	0.226	0.204	0.226	0.204

Table B.3: Table that shows the precision of the rules trained on the family on the row with the parameters indicated by the respective column. All the rules used for the tests have been generated using the Russell Rao distance.

Family	Size	hdbscan	hdbscan+rules	hdbscan+goodware	hdbscan+rules+ goodware	hdbscan:clot	hdbscan:clot+rules	hdbscan:clot+ goodware	hdbscan:clot+rules+ goodware
agentb	3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
allaple	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
atraps	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
aura	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
autoit	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
barys	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
beebone	3	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
bitman	1090	0.807	0.812	0.807	0.812	0.807	0.812	0.807	0.812
bunitu	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
carberp	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
cerber	533	0.912	0.911	0.912	0.911	0.912	0.911	0.912	0.911
cloud	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
coantor	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
critroni	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Continued on next page

Table B.4 – continued from previous page

Family	Size	hdbscan	hdbscan+rules	hdbscan+goodware	hdbscan+rules+ goodware	hdbscan:clot	hdbscan:clot+rules	hdbscan:clot+ goodware	hdbscan:clot+rules+ goodware
crowti	75	0.264	0.297	0.78	0.78	0.139	0.298	0.767	0.767
crypmod	42	0.066	0.067	0.066	0.067	0.066	0.067	0.066	0.067
cryptowall	47	0.184	0.152	0.194	0.152	0.125	0.152	0.194	0.152
dagozill	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
dalexis	8	0.212	0.209	0.212	0.209	0.75	0.357	0.75	0.357
delf	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
deshacop	8	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
dmalocker	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dridex	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dynamer	3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
enestaller	3	0.778	0.778	0.778	0.778	0.778	0.778	0.778	0.778
enestedel	6	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
fareit	14	0.5	1.0	0.5	1.0	0.5	1.0	0.5	1.0
fraudrop	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
gamarue	33	0.1	0.143	0.1	0.143	0.1	0.143	0.1	0.143
genericcryptor	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
genkryptik	3	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
glupteba	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
hplocky	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
lethic	28	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
locky	647	0.263	0.273	0.263	0.273	0.263	0.04	0.263	0.264
midie	6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
mikey	8	0.028	0.032	0.028	0.032	0.042	0.054	0.042	0.054
myxah	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ngrbot	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
onion	40	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rack	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
razy	40	0.595	0.587	0.595	0.587	0.595	0.587	0.595	0.587
reconyc	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ruskill	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Continued on next page

Table B.4 – continued from previous page

Family	Size	hdbscan	hdbscan+rules	hdbscan+goodware	hdbscan+rules+ goodware	hdbscan:clot	hdbscan:clot+rules	hdbscan:clot+ goodware	hdbscan:clot+rules+ goodware
sage	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
sagecrypt	47	0.608	0.66	0.608	0.66	0.608	0.66	0.608	0.66
scar	25	0.457	0.457	0.457	0.457	0.457	0.457	0.457	0.457
scatter	12	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
score	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
shade	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
shiz	104	0.781	0.897	0.781	0.897	0.781	0.897	0.781	0.897
teerac	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
tescrypt	29	0.412	0.471	0.455	0.455	0.412	0.471	0.455	0.455
teslacrypt	2478	0.9	0.893	0.9	0.893	0.9	0.893	0.9	0.893
tinba	7	0.111	0.111	0.111	0.111	0.111	0.111	0.111	0.111
torrentlocker	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
tpyn	13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
upatre	154	0.167	0.125	0.205	0.4	0.167	0.125	0.205	0.4
waldek	7	0.5	0.5	0.5	0.5	NaN	0.5	NaN	0.5
yakes	682	0.414	0.446	0.414	0.446	0.414	0.446	0.414	0.446
zbot	43	0.714	0.714	0.714	0.714	0.714	0.714	0.714	0.714
zboter	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
zegost	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
zerber	329	0.692	0.683	0.692	0.683	0.692	0.683	0.692	0.683
zusy	65	0.174	0.215	0.174	0.215	0.171	0.215	0.171	0.215

Table B.4: Table that shows the precision of the rules trained on the family on the row with the parameters indicated by the respective column. All the rules used for the tests have been generated using the Jaccard distance.

B.2 yarGen Precision tables

Family	**Size**	**plain**	**goodware**	**excludegood**	**goodware+ excludegood**	**opcodes**	**opcodes+ goodware**	**opcodes+ excludegood**	**opcodes+ goodware+ excludegood**
agentb	3	0.083	0.0	0.083	0.0	0.083	0.0	0.083	0.0
allaple	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
atraps	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
aura	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
autoit	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
barys	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
beebone	3	0.5	0.5	0.5	0.5	1.0	1.0	1.0	1.0
bitman	1090	0.625	0.69	0.625	0.69	0.795	0.852	0.795	0.852
bunitu	2	0.308	0.308	0.308	0.308	0.5	0.5	0.5	0.5
carberp	2	0.0	0.0	0.0	0.0	NaN	NaN	NaN	NaN
cerber	533	0.622	0.657	0.622	0.657	0.704	0.74	0.704	0.74
cloud	2	0.061	0.061	0.061	0.061	0.375	0.375	0.375	0.375
coantor	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
critroni	3	0.25	0.25	0.25	0.25	NaN	NaN	NaN	NaN
crowti	75	0.468	0.275	0.468	0.275	0.605	0.615	0.605	0.615
crypmod	42	0.071	0.084	0.071	0.084	0.133	0.167	0.133	0.167
cryptowall	47	0.125	0.195	0.125	0.195	0.636	0.273	0.636	0.273
dagozill	4	0.125	0.25	0.125	0.25	0.0	0.0	0.0	0.0
dalexis	8	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
delf	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
deshacop	8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dmalocker	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dridex	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dynamer	3	0.011	0.02	0.011	0.02	0.0	0.02	0.0	0.02
enestaller	3	0.2	0.2	0.2	0.2	0.444	0.444	0.444	0.444
enestedel	6	0.192	0.179	0.192	0.179	0.192	0.192	0.192	0.192
fareit	14	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
fraudrop	3	0.044	0.044	0.044	0.044	0.5	0.5	0.5	0.5
gamarue	33	0.086	0.037	0.086	0.037	0.2	0.333	0.2	0.333
genericcryptor	2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
genkryptik	3	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
glupteba	4	0.015	0.015	0.015	0.015	0.026	0.026	0.026	0.026
hplocky	2	0.667	0.667	0.667	0.667	0.667	0.667	0.667	0.667
lethic	28	0.0	0.048	0.0	0.048	NaN	1.0	NaN	1.0

Continued on next page

Table B.5 – continued from previous page

Family	**Size**	**plain**	**goodware**	**excludegood**	**goodware+ excludegood**	**opcodes**	**opcodes+ goodware**	**opcodes+ excludegood**	**opcodes+ goodware+ excludegood**
locky	647	0.048	0.044	0.048	0.044	0.14	0.246	0.14	0.246
midie	6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
mikey	8	0.075	0.357	0.075	0.357	0.0	0.0	0.0	0.0
myxah	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ngrbot	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
onion	40	0.026	0.026	0.026	0.026	0.029	0.029	0.029	0.029
rack	3	0.034	0.034	0.034	0.034	0.167	0.167	0.167	0.167
razy	40	0.673	0.638	0.673	0.638	0.328	0.267	0.328	0.267
reconyc	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
ruskill	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
sage	4	0.154	0.154	0.154	0.154	0.154	0.154	0.154	0.154
sagecrypt	47	0.526	0.526	0.526	0.526	0.526	0.526	0.526	0.526
scar	25	0.471	0.471	0.471	0.471	1.0	1.0	1.0	1.0
scatter	12	0.032	0.009	0.032	0.009	0.027	0.009	0.027	0.009
score	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
shade	2	0.011	0.006	0.011	0.006	0.016	0.006	0.016	0.006
shiz	104	0.908	0.908	0.908	0.908	0.908	0.908	0.908	0.908
teerac	3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
tescrypt	29	0.369	0.433	0.369	0.433	0.333	0.472	0.333	0.472
teslacrypt	2478	0.886	0.886	0.886	0.886	0.922	0.922	0.922	0.922
tinba	7	0.0	0.111	0.0	0.111	0.0	0.111	0.0	0.111
torrentlocker	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
tpyn	13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
upatre	154	0.161	0.118	0.161	0.118	0.346	0.323	0.346	0.323
waldek	7	NaN	0.0	NaN	0.0	NaN	NaN	NaN	NaN
yakes	682	0.466	0.466	0.466	0.466	0.474	0.474	0.474	0.474
zbot	43	NaN	0.333	NaN	0.333	NaN	1.0	NaN	1.0
zboter	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
zegost	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
zerber	329	0.526	0.513	0.526	0.513	0.64	0.507	0.64	0.507
zusy	65	0.091	0.067	0.091	0.067	0.22	0.189	0.22	0.189

Continued on next page

Table B.5 – continued from previous page

Family	Size	plain	goodware	excludegood	goodware+ excludegood	opcodes	opcodes+ goodware	opcodes+ excludegood	opcodes+ goodware+ excludegood

Table B.5: Table that shows the precision obtained by the yarGen generated rules over the family indicated in each row and using the parameters in each column.

Family	Size	plain	goodware	excludegood	goodware+ excludegood	opcodes	opcodes+ goodware	opcodes+ excludegood	opcodes+ goodware+ excludegood
agentb	3	0.077	0.077	0.077	0.077	0.077	0.077	0.077	0.077
allaple	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
atraps	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
aura	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
autoit	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
barys	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
beebone	3	0.5	0.5	0.5	0.5	1.0	1.0	1.0	1.0
bitman	1090	0.786	0.667	0.81	0.684	0.837	0.809	0.835	0.839
bunitu	2	NaN	1.0	NaN	NaN	NaN	1.0	NaN	NaN
carberp	2	0.0	0.0	0.0	0.0	NaN	NaN	NaN	NaN
cerber	533	0.698	0.591	0.719	0.757	0.827	0.802	0.83	0.765
cloud	2	0.375	0.375	0.375	0.375	0.375	0.375	0.375	0.375
coantor	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
critroni	3	0.333	0.333	0.333	0.25	0.5	0.5	0.5	NaN
crowti	75	0.632	0.623	0.627	0.623	0.648	0.648	0.648	0.648
crypmod	42	0.113	0.106	0.113	0.107	0.174	0.167	0.174	0.172
cryptowall	47	0.229	0.183	0.273	0.273	0.41	0.439	0.258	0.258
dagozill	4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
dalexis	8	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
delf	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
deshacop	8	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dmalocker	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dridex	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
dynamer	3	0.0	0.012	0.0	0.021	0.0	0.012	0.0	0.021

Continued on next page

Table B.6 – continued from previous page

Family	Size	plain	goodware	excludegood	goodware+ excludegood	opcodes	opcodes+ goodware	opcodes+ excludegood	opcodes+ goodware+ excludegood
enestaller	3	0.2	0.2	0.2	0.2	0.444	0.444	0.444	0.444
enestedel	6	0.192	0.192	0.192	0.192	0.192	0.192	0.192	0.192
fareit	14	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
fraudrop	3	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
gamarue	33	0.09	0.062	0.059	0.065	0.111	0.115	0.114	0.114
genericcryptor	2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
genkryptik	3	0.25	0.25	0.25	0.25	0.25	0.25	0.25	0.25
glupteba	4	0.25	0.025	0.25	0.025	0.026	0.026	0.026	0.026
hplocky	2	0.667	0.667	0.667	0.667	0.667	0.667	0.667	0.667
lethic	28	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
locky	647	0.15	0.042	0.232	0.043	0.236	0.076	0.241	0.07
midie	6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
mikey	8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
myxah	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ngrbot	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
onion	40	0.014	0.01	0.026	0.026	0.029	0.083	0.029	0.029
rack	3	0.034	0.034	0.034	0.034	0.167	0.167	0.167	0.167
razy	40	0.56	0.489	0.49	0.5	0.297	0.297	0.297	0.297
reconyc	3	NaN	NaN	NaN	NaN	NaN	0.0	NaN	NaN
ruskill	2	NaN	0.0	NaN	0.0	NaN	NaN	NaN	NaN
sage	4	0.154	0.154	0.154	0.154	0.154	0.154	0.154	0.154
sagecrypt	47	0.345	0.345	0.541	0.541	0.541	0.541	0.541	0.541
scar	25	0.81	0.81	0.81	0.81	1.0	0.895	1.0	0.895
scatter	12	0.02	0.019	0.026	0.03	0.027	0.018	0.027	0.031
score	2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
shade	2	0.014	0.007	0.014	NaN	0.016	0.007	0.016	NaN
shiz	104	0.908	0.908	0.908	0.908	0.908	0.908	0.908	0.908
teerac	3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
tescrypt	29	0.352	0.402	0.453	0.373	0.415	0.435	0.415	0.415
teslacrypt	2478	0.805	0.805	0.882	0.882	0.906	0.906	0.905	0.905
tinba	7	0.111	0.019	0.111	0.111	0.111	0.111	0.111	0.111
torrentlocker	2	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
tpyn	13	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Continued on next page

Table B.6 – continued from previous page

Family	**Size**	**plain**	**goodware**	**excludegood**	**goodware+ excludegood**	**opcodes**	**opcodes+ goodware**	**opcodes+ excludegood**	**opcodes+ goodware+ excludegood**
upatre	154	0.146	0.095	0.273	0.161	0.306	0.297	0.3	0.281
waldek	7	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
yakes	682	0.405	0.405	0.471	0.471	0.482	0.482	0.482	0.482
zbot	43	0.182	0.182	0.1	0.182	0.182	0.182	0.1	0.182
zboter	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
zegost	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
zerber	329	0.674	0.693	0.66	0.76	0.681	0.677	0.673	0.785
zusy	65	0.191	0.15	0.215	0.203	0.207	0.207	0.208	0.207

Table B.6: Table that shows the precision obtained by the yarGen generated rules over the family indicated in each row and using the parameters in each column. As opposed to Table 4.19, all the trainings have been performed by using the "z0" parameter together with the indicated ones.

Bibliography

[1] Andrea Atzeni et al. “Countering Android Malware: a Scalable Semi-Supervised Approach for Family-Signature Generation”. In: *IEEE Access* (2018).

[2] AV-Test. *Malware Statistics*. https://www.av-test.org/en/statistics/malware/.

[3] AV-TEST. *SECURITY REPORT 2017/18*. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2017-2018.pdf.

[4] *Awesome Yara Rules*. https://github.com/InQuest/awesome-yara#rules.

[5] Dixon B. *Watching Attackers Through VirusTotal*. http://blog.9bplus.com/watching-attackers-through-virustotal/. Sept. 1, 2014.

[6] Jayanta Basak and Raghu Krishnapuram. “Interpretable hierarchical clustering by constructing an unsupervised decision tree”. In: *IEEE transactions on knowledge and data engineering* 17.1 (2005), pp. 121–132.

[7] *BASS - BASS Automated Signature Synthesizer*. https://github.com/Cisco-Talos/BASS.

[8] Rankin Bert. *A Brief History of Malware — Its Evolution and Impact*. https://www.lastline.com/blog/history-of-malware-its-evolution-and-impact/.

[9] Battista Biggio et al. “Evasion attacks against machine learning at test time”. In: *Joint European conference on machine learning and knowledge discovery in databases*. Springer. 2013, pp. 387–402.

[10] Clark C. *yaraGenerator*. https://github.com/Xen0ph0n/YaraGenerator. 2013.

[11] Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. “Density-based clustering based on hierarchical density estimates”. In: *Pacific-Asia conference on knowledge discovery and data mining*. Springer. 2013, pp. 160–172.

[12] Ricardo JGB Campello et al. “Hierarchical density estimates for data clustering, visualization, and outlier detection”. In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 10.1 (2015), p. 5.

[13] Ero Carrera. *pefile*. https://github.com/erocarrera/pefile. 2018.

[14] Douglas McKee Charles Crofford. *Ransomware Families Use NSIS Installers to Avoid Detection, Analysis.* https://securingtomorrow.mcafee.com/mcafee-labs/ransomware-families-use-nsis-installers-to-avoid-detection-analysis/. Mar. 28, 2017.

[15] Marie Chavent. "A monothetic clustering method". In: *Pattern Recognition Letters* 19.11 (1998), pp. 989–996.

[16] Marie Chavent, Yves Lechevallier, and Olivier Briant. "DIVCLUS-T: A monothetic divisive hierarchical clustering method". In: *Computational Statistics & Data Analysis* 52.2 (2007), pp. 687–701.

[17] Thomas M Chen and Jean-Marc Robert. "The evolution of viruses and worms". In: *Statistical methods in computer security* 1 (2004).

[18] Seung-Seok Choi, Sung-Hyuk Cha, and Charles C Tappert. "A survey of binary similarity and distance measures". In: *Journal of Systemics, Cybernetics and Informatics* 8.1 (2010), pp. 43–48.

[19] Hopping Clare. *Carbon Black: 92% of businesses have been attacked in the last 12 months.* http://www.itpro.co.uk/cyber-attacks/31893/carbon-black-92-of-businesses-have-been-attacked-in-the-last-12-months. Sept. 12, 2018.

[20] Mojtaba Eskandari, Zeinab Khorshidpour, and Sattar Hashemi. "HDM-Analyser: a hybrid analysis approach based on data mining techniques for malware detection". In: *Journal of Computer Virology and Hacking Techniques* 9.2 (2013), pp. 77–93.

[21] Martin Ester. "Density-based clustering". In: *Encyclopedia of Database Systems.* Springer, 2009, pp. 795–799.

[22] Roth F. *yarGen.* https://github.com/Neo23x0/yarGen. 2018.

[23] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. "Malware analysis and classification: A survey". In: *Journal of Information Security* 5.02 (2014), p. 56.

[24] Mariano Graziano et al. "Needles in a haystack: mining information from public dynamic analysis sandboxes for malware intelligence". In: *USENIX Security Symposium.* USENIX Association. 2015, pp. 1057–1072.

[25] GReAT. *The devil's in the Rich header.* https://securelist.com/the-devils-in-the-rich-header/84348/. Mar. 8, 2018.

[26] Trend Micro Inc. *Cryptocurrency-Mining Malware: 2018's New Menace?* https://blog.trendmicro.com/trendlabs-security-intelligence/cryptocurrency-mining-malware-2018-new-menace/.

[27] Trend Micro Inc. *Ransomware.* https://www.trendmicro.com/vinfo/us/security/definition/Ransomware.

[28] Zetter K. *A google site meant to protect you is helping hackers attack you.* http://www.wired.com/2014/09/how-hackers-use-virustotal/. 2014.

[29] Kaspersky. *Antivirus fundamentals: Viruses, signatures, disinfection.* https://www.kaspersky.com/blog/signature-virus-disinfection/13233/.

[30] Kaspersky. *Multipurpose malware: Sometimes Trojans come in threes.* https://www.kaspersky.com/blog/cerber-multipurpose-malware/12221.

[31] Kaspersky. *TeslaCrypt Ransomware Attacks.* https://www.kaspersky.com/resource-center/threats/teslacrypt.

[32] Jesse Kornblum. "Identifying almost identical files using context triggered piecewise hashing". In: *Digital investigation* 3 (2006), pp. 91–97.

[33] Kaspersky Lab. *History of malicious programs.* https://securelist.com/threats/history-of-malicious-programs/.

[34] Mandiant. *Tracking Malware with Import Hashing.* https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html. Jan. 23, 2014.

[35] Andrea Marcelli and Giovanni Squillero. *YaYaGen - Yet Another Yara Rule Generator.* https://github.com/jimmy-sonny/YaYaGen.

[36] Leland McInnes and John Healy. "Accelerated Hierarchical Density Clustering". In: *arXiv preprint arXiv:1705.07321* (2017).

[37] Sikorski Michale and Honig Andrew. *Practical Malware Analysis - The Hands-On Guide to Dissecting Malicious Software.* No Starch Press, 2012, pp. 3–4.

[38] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Limits of static analysis for malware detection". In: *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual.* IEEE. 2007, pp. 421–430.

[39] AlienVault OTX. *Yabin.* https://github.com/AlienVault-OTX/yabin. 2018.

[40] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. "Transferability in machine learning: from phenomena to black-box attacks using adversarial samples". In: *arXiv preprint arXiv:1605.07277* (2016).

[41] Matt Pietrek. *Peering Inside the PE: A Tour of the Win32 Portable Executable File Format.* https://msdn.microsoft.com/en-us/library/ms809762.aspx. 1994.

[42] Daniel Pistelli. *Microsoft's Rich Signature (undocumented).* http://www.ntcore.com/files/richsign.htm. Nov. 11, 2010.

[43] Wesson R. *Project Icewater.* https://github.com/SupportIntelligence/Icewater. 2018.

[44] Perez Roi. *Microsoft says Cerber ransomware most popular infector of Windows 10.* https://www.scmagazineuk.com/microsoft-says-cerber-ransomware-popular-infector-windows-10/article/1475374. Feb. 1, 2017.

[45] Andrew Rosenberg and Julia Hirschberg. "V-measure: A conditional entropy-based external cluster evaluation measure". In: *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*. 2007.

[46] Kevin A Roundy and Barton P Miller. "Binary-code obfuscations in prevalent packer tools". In: *ACM Computing Surveys (CSUR)* 46.1 (2013), p. 4.

[47] Kevin A Roundy and Barton P Miller. "Hybrid analysis and control of malware". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2010, pp. 317–338.

[48] Mostafa E Saleh, A Baith Mohamed, and A Abdel Nabi. "Eigenviruses for metamorphic virus recognition". In: *IET information security* 5.4 (2011), pp. 191–198.

[49] Mike Schiffman. *A Brief History of Malware Obfuscation: Part 1 of 2*. https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2. 2010.

[50] Mike Schiffman. *A Brief History of Malware Obfuscation: Part 2 of 2*. https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_2_of_2. 2010.

[51] Matthew G Schultz et al. "Data mining methods for detection of new malicious executables". In: *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*. IEEE. 2001, pp. 38–49.

[52] Marcos Sebastián et al. "Avclass: A tool for massive malware labeling". In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer. 2016, pp. 230–253.

[53] *Set cover problem*. https://en.wikipedia.org/wiki/Set_cover_problem.

[54] M Zubair Shafiq, S Tabish, and Muddassar Farooq. "PE-probe: leveraging packer detection and structural information to detect malicious portable executables". In: *Proceedings of the Virus Bulletin Conference (VB)*. Vol. 8. 2009.

[55] M Zubair Shafiq et al. "Pe-miner: Mining structural information to detect malicious executables in realtime". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2009, pp. 121–141.

[56] *Simile (computer virus)*. https://en.wikipedia.org/wiki/Simile_(computer_virus).

[57] Arne Swinnen and Alaeddine Mesbahi. "One packer to rule them all: Empirical identification, comparison and circumvention of current Antivirus detection techniques". In: *BlackHat USA* (2014).

[58] *The hdbscan Clustering Library*. https://hdbscan.readthedocs.io/en/latest/.

[59] *The Kam1n0 Assembly Analysis Platform.* https://github.com/McGill-DMaS/Kam1n0-Community.

[60] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. "Survey on the usage of machine learning techniques for malware analysis". In: *arXiv preprint arXiv:1710.08189* (2017).

[61] Xabier Ugarte-Pedrero et al. "SoK: Deep packer inspection: A longitudinal study of the complexity of run-time packers". In: *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2015, pp. 659–673.

[62] Dolly Uppal, Vishakha Mehra, and Vinod Verma. "Basic survey on malware analysis, tools and techniques". In: *International Journal on Computational Sciences & Applications (IJCSA) Vol* 4 (2014), pp. 103–111.

[63] *UPX - the Ultimate Packer for eXecutables.* https://github.com/upx/upx.

[64] *VirusTotal.* https://www.virustotal.com/.

[65] *Writing YARA rules.* https://yara.readthedocs.io/en/v3.7.0/writingrules.html.

[66] *x86 Disassembly/Windows Executable Files.* https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files#Resources.

[67] *YARA - The pattern matching swiss knife.* https://github.com/VirusTotal/yara.

[68] *YARA. The pattern matching swiss knife.* https://github.com/VirusTotal/yara.

[69] Ilsun You and Kangbin Yim. "Malware obfuscation techniques: A brief survey". In: *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*. IEEE. 2010, pp. 297–300.