Apìcula: Static Detection of API Calls in Generic Streams of Bytes

Mario D'Onghia, Matteo Salvadore, Benedetto Maria Nespoli, Michele Carminati, Mario Polino, Stefano Zanero[a]

[a]*Dipartimento di Elettronica, Informazione e Bioingengeria, Milan, Italy*

Abstract

API functions often require the crafting of specific inputs and may return some output that is usually processed by the code that immediately follows their invocation. In this work, we claim that - for *some* APIs - those two stages are both frequently similar across different binaries and sufficiently unique to be fingerprinted.

We build upon this intuition and present *Apìcula*, a static analysis tool for identifying API calls in generic streams of bytes, such as memory dumps, network traffic, or object code files, for which known techniques do not suffice. In a nutshell, Apìcula leverages the control flow graph of a binary to generate a set of fingerprints for all basic blocks that end with a call instruction; those sets are then compared against a database of pre-computed fingerprints to establish whether any known API is being invoked.

We provide a series of experiments that are instrumental (1) in demonstrating that the same fingerprints computed for specific APIs can be observed across different binaries and (2) in identifying a subset of the Windows APIs whose usage can be detected by Apìcula with sufficient precision and sensitivity, focusing in particular on malicious binaries. Furthermore, we illustrate two techniques that can be used to validate different fingerprint databases in case someone wants to detect APIs belonging to libraries different from those that we consider in this work.

In particular, we prove that fingerprints associated with different APIs are remarkably dissimilar and therefore can be employed for distinguishing between APIs. More specifically, we find that fingerprint sets associated with *different* APIs present on average a Jaccard index value of 0.000125; in comparison, the average similarity between fingerprint sets associated with the *same* API is 0.29 (Jaccard index) for binaries compiled with the same optimization level and 0.07 (Jaccard index) for binaries compiled with different optimization levels. Moreover, we show that we can build databases of fingerprints that are sufficiently comprehensive to identify specific APIs in unseen binaries. More precisely, we identify 228 different APIs among the Windows APIs (including the C run-time libraries) whose usage can be detected by Apìcula with sensitivity greater than 80% and a false discovery rate lower than 5%.

Keywords: Application Programming Interface Detection, Binary Analysis, Static Analysis, Code Fingerprinting, Malware Analysis

Email address: `{mario.donghia, michele.carminati,mario.polino,stefano.zanero}@polimi.it`, `{matteo.salvadore, benedettomaria.nespoli}@mail.polimi.it` (Mario D'Onghia, Matteo Salvadore, Benedetto Maria Nespoli, Michele Carminati, Mario Polino, Stefano Zanero)

1. Introduction

Due to the semantic meaning they bear, as well as the high volume of information that can be indirectly inferred from them, Application Programming Interfaces (APIs) are of crucial importance in both the detection and analysis of malware. For instance, APIs have been used for classification purposes (i.e., the attribution of malware samples to a specific family or type), either cumulatively as sequences of system calls that define a specific *behavior* [1, 2, 3] or individually, when their mere frequency or the arguments that are passed to them as inputs can be indicative of potentially malicious activity by a program [4, 5]. Furthermore, identifying APIs in a given executable is a pivotal step for analysts wishing to derive how that program works [6, Chapter 9], as well as for estimating the consequences of a malware attack as part of the forensics tasks that are usually conducted after a cyber-incident (e.g., Li *et al.* [7] proposed systematic procedures for reconstructing malicious events through code analysis on Android devices).

Sequences of API calls (alternatively known as API call *traces*) are usually collected by letting a sample execute in a controlled environment, such as a sandbox, which is instrumented to record the activity of a malicious sample, including all the API calls it performs. This type of approach, which is usually referred to as *dynamic analysis*, can facilitate the understanding of a program's functions and goals, especially when this employs obfuscation techniques to hinder code analysis. However, dynamic analysis can be highly resource-demanding, and it is often incapable of producing an exhaustive report for a given program since, in most cases, only some of its branches are executed [8, 9]. On top of that, this class of techniques requires samples to be *working* executables, which makes them unemployable in some rather important forensics tasks, such as the analysis of memory dumps.

On the other hand, static analysis techniques do not require the direct observation of a program's execution and ideally allow for a more comprehensive analysis, albeit being quite fragile against obfuscation techniques and almost powerless against packed malware (Moser *et al.* [10] demonstrates that de-obfuscating malware can be as hard as solving the 3SAT problem). However, if we aim at identifying API calls in executable code, we can analyze some of the fields contained in the headers of the executable (such as the Import Address Table for Windows PEs and the Procedure Linkage Table for Linux ELFs) and use that information to reconstruct which API call was made and where it was performed in the execution flow of the program [11]. In an ideal setting, that would be sufficient to identify almost every API call in a binary program.

In this work, we tackle a more challenging scenario, in which we cannot assume to be able to locate API calls by simply looking at the information contained within the headers of an executable: in fact, we do not assume to be dealing with an executable program at all; instead, our goal is to discover API calls in generic streams of bytes, such as memory dumps, object code files, or network traffic. We aim to do that by fingerprinting the code that *surrounds* specific API calls; we then use those fingerprints to identify the corresponding APIs in previously unseen streams of bytes. Our intuition is based on the fact that *some* APIs require the crafting of special inputs and may also return some output which is usually processed within the code that immediately follows the API invocation. We claim that, for some APIs, the stages of input preparation and output processing are somewhat similar across different programs and, at the same time, sufficiently unique not to be confused with those of other APIs. We implement this intuition in a static analysis tool that we have named *Apìcula*. In a nutshell, Apìcula first generates a set of fingerprints for each *call* instruction found within the byte stream; then, it compares those sets against a database of fingerprints that were previously observed for the APIs that we want to detect. In particular, if

Apìcula finds that the matching rate for a given call instruction and a specific API is greater than a fixed threshold, then it infers that the call instruction is actually invoking that API.

We evaluate our system with a series of experiments that are primarily intended to verify that Apìcula can distinguish among different APIs with high accuracy. In particular, we observe how fingerprint sets associated with *different* APIs share on average a very low similarity rate (average Jaccard index 0.000125), whereas fingerprint sets associated with the *same* API are significantly more similar (average Jaccard index 0.29 when compiling binaries with the same optimization level, 0.07 when compiling with different ones). Moreover, by testing our system against a dataset of real-world malicious binaries for the Windows operating system, we are able to produce a list of 228 APIs among the Windows APIs, including the C run-time library, that Apìcula can detect with sufficient sensitivity (i.e., greater than 80%) and negligible false discovery rate (i.e., lower than 5%).

In summary, we make the following novel contributions:

1. we provide a new method for fingerprinting call instructions in executable code, with particular emphasis on the *x86* architecture;
2. we show that, for some API functions, the code sections that precede and follow their invocation are similar across different binaries;
3. we demonstrate that, for some API functions, we can build an exhaustive database of fingerprints that can be used to detect the usage of those APIs in unknown streams of bytes;
4. lastly, we produce a list of the Windows APIs whose usage that can be detected with sufficient precision and sensitivity.

The remainder of this paper is structured as follows: first, we mention some of the most relevant works in code and function matching, highlighting the main differences with Apìcula; next, we provide an exhaustive description of Apìcula, its main components, as well as the design choices that were made in order to maximize its accuracy; we then describe a series of experiments that we produced in order to prove that Apìcula actually works, as well as to calibrate some parameters that Apìcula requires in order to function; moreover, we provide two strategies that can be used to easily assess the quality of a newly constructed database of fingerprints; lastly, we discuss the limitations and potential applications of the tool.

2. Related Work

There has been growing interest in automated reverse engineering research in recent years, particularly in the automatic identification of functions in assembly code. Ding et al. [12] presented a learning model based on assembly code representation, which was shown to be efficacious in identifying functions within executable code, including API functions. Similarly, the same authors [13] first introduced an identification approach that combines *locality-sensitive hashing* and sub-graph search to address the problem of scalability, which affected previous works. Polino et al. [14] used fingerprinting to cluster similar behaviors, but the semantic of those behaviors was extracted using a dynamic execution approach. On a related note, [15, 16, 17, 18, 19] are all other examples of search engines that exploit code similarity to identify known functions and sections of code. Moreover, graph isomorphism has been widely employed to determine the similarity between binaries [20, 21, 22].

All of the above works exploit the fact that code re-use is so frequent that different binaries may share similar parts or characteristics. This is not far from what we attempt to achieve with

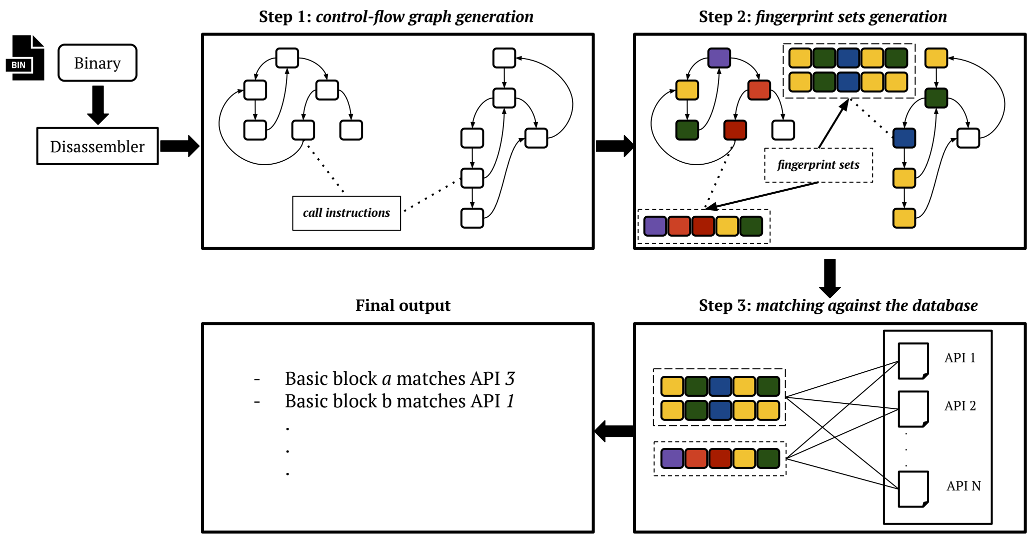
Step 1: *control-flow graph generation*
Step 2: *fingerprint sets generation*
Binary
BIN
Disassembler
call instructions
fingerprint sets
Final output
Step 3: *matching against the database*
- Basic block *a* matches API *3*
- Basic block b matches API *1*
API 1
API 2
API N

Figure 1: The three stages that compose the Apìcula workflow.

Apìcula, although we believe that the efficaciousness of our system is not exclusively due to code re-use: on the contrary, we strongly believe that the code surrounding certain APIs is intrinsically similar across different binaries due to the specific input preparation and output handling, which, respectively, is required by and comes with using those APIs.

Another major difference lies in the overall goal of Apìcula when compared against other published works, as we try neither to determine the similarity among different binaries nor to identify any particular code section within a binary. For instance, Ding et al. [12] focus on identifying the *actual* code of a function, including that of known API functions. Similarly, the approach developed by De Nicolao et al. [23] identifies the architecture and differentiate between code and data. However, they do not address the problem of identifying the *usage* of APIs in executable code, which is what we attempt to achieve in this work.

3. Approach

Apìcula identifies API calls in binaries by exploiting a sets of pre-computed fingerprints. This section provides the details on how Apìcula works internally, especially on how the fingerprint sets are first generated and then compared against each other.

3.1. Apìcula Overview

As shown in Figure 1, the work-flow of Apìcula is composed of three distinct and consecutive stages, namely:

1. *disassembling* and *control flow graph* generation;
2. *fingerprint sets generation*;
3. *matching* against the database.

In summary, Apìcula works as follows: on input a byte stream, Apìcula disassembles it and constructs the corresponding control-flow graph, keeping track of the basic blocks that end with a call instruction (*1. disassembling and control flow graph generation*); next, it generates a set of fingerprints for each of those blocks (*2. fingerprint sets generation*), which are then compared against a database of previously observed fingerprints (*3. matching against the database*); in particular, this comparison is carried out by means of the overlap coefficient, which is used to establish whether a given block is invoking any of the APIs that we want to detect.

3.2. Disassembling and Control Flow Graph Generation

The first essential task that Apìcula needs to perform, in order to compute and match fingerprints, is the disassembling of the binary and the re-organization of the code in a control flow graph structure. A control flow graph (CFG in short from now on) is a directed graph in which each node, also known as a *basic block*, represents a linear sequence of program instructions that are always executed in sequence, whereas the edges represent control flow paths [24]. CFGs are often employed in tasks such as compiler optimization and program analysis, including malware. In Apìcula, we need to re-organize the binary under analysis in a CFG because, as it will be discussed thoroughly in the next sections, our fingerprints are essentially sequences of *colored* basic blocks extracted from the computational paths that pass through a node containing a *call* instruction.

3.2.1. Code Disassembling

Before a CFG can be built, Apìcula needs to disassemble the byte stream: we use *Capstone*, a lightweight disassembler supporting several architectures and instruction sets [25]. It is important to notice that, since our inputs are general byte streams and not necessarily binaries in some well-known executable format (e.g., PE32), the initial bytes in the stream might be corrupted or incomplete or contain no instructions whatsoever, thus jeopardizing the ultimate correctness of the entire disassembling process. However, some instruction set architectures, including the widely used *x86*, possess the property of being *self-repairing*, namely, regardless of any error that may happen during the disassembling of the byte stream, the disassembler *eventually* ends up re-synchronizing with the correct stream of instructions [26], thus mitigating the aforementioned problem. In the current version of Apìcula, we focused exclusively on the *x86* instruction set architecture. However, it could be easily extended for any other architecture, as long as it maintains the property of being self-repairing. Finally, during the disassembling process, we do not parse or analyze the operands of any of the instructions, with the sole exception of the landing addresses of *jump* and *call* instructions, which are needed in order to construct the CFG.

3.2.2. CFG Generation

The generation of the CFG is composed by the following two stages: (1) basic block generation and (2) basic block linking.

Basic block generation. The byte stream is scanned, and the addresses of the instructions that start new basic blocks are identified and stored in a list. The byte stream is divided into basic blocks, each of them starting at one of the identified addresses; furthermore, blocks that end with a *call* instruction are marked accordingly and a reference to them is stored in a separate data structure.

Basic block linking. Basic blocks are linked together by checking their last instruction. In case the last instruction is:

- a *return* or *halt* instruction, the block does not get connected to any child blocks;

- an *unconditional jump*, then the block is linked to the block that starts at the jump address if that exists;
- a *call* to an internal subroutine (i.e., that can be found within the byte stream) or a *conditional jump* instruction, the block is linked to the one that starts at the call/jump address, if that exists, as well as to the block whose starting address immediately follows the end address of the block we are linking. In other words, the second block is the one that contains the code that will be executed if the jump is not taken or where the execution flow will continue after the subroutine that is invoked through the call instruction has returned;
- a *call* instruction to an address location, which is *external* to the address offset of the byte stream, the block is linked to the block whose start address immediately follows. We are particularly interested in this type of nodes because those are the basic blocks that may invoke one of the APIs that we are trying to detect with Apìcula.

It is also important to point out that, during the linking phase, we do not just link one node to its children, but we also link the children to their parents: we require this because, in order to fingerprint a block, we must also visit the graph backward.

3.3. Fingerprinting

The second phase in the Apìcula work-flow is the computation of a fingerprint set for each of the basic blocks that contain a *call* instruction to an *external*[1] address. For a given basic block b, each fingerprint represents a computational path that starts before b, passes through it, and eventually continues ahead, incorporating a certain number of blocks reachable from b. Ideally, b is the basic block containing a call instruction to an API, whereas the blocks that precede it (and from which there exists a path that leads to it) are the ones responsible for preparing and pushing onto the stack the inputs to the API function. Finally, the blocks that follow b are the ones in which the program processes the potential output of the API function. Given that fingerprints are sequences of basic blocks of the CFG, we need a method for encoding them so that the resulting fingerprints can be compared between each other. We borrowed from Kruegel et al. [22] the concept of *color* as a means for representing the content of a node in a compact and meaningful way. A color is an array of bits, where each bit corresponds to a certain class of instructions. In this array, the i^{th} bit is set to 1 *iff* the block contains at least an instruction belonging to the i^{th} instruction class, to 0 otherwise, with the only exception of the very first bit of the array, which we reserve for encoding special symbols that we need for separating the fingerprints inside the database. Table 1 summarizes the instruction classes that we have included in Apìcula.

In order to be comparable among them, all fingerprints must have a fixed length. We considered the possibility of assigning a different fingerprint length for each API. However, we eventually decided to maintain a consistent length for all of them due to the high overhead and the additional complexity that this would add to the system. In Section 4.2 we present an experiment that we conducted to try and identify an optimal length for our fingerprints. Nonetheless, we still leave the possibility of specifying a custom length to the final users since they may want to use Apìcula on different API sets, for which the optimal length might be different. In general, not hard-coding a fixed length can allow the system to be more flexible in case some malicious sample

[1]as in "not contained within the address offset of the byte stream".

Instruction Classes		
Bit	Class Name	Examples
1	NO CLASS	nop
2	ARITHMETIC	add, div, idiv
3	BRANCH	jne, jz, loop
4	CALL	call, lcall
5	CONDITIONAL MOVE	cmova, cmove
6	DATA TRANSFER	mov, movs
7	FLAGS	bt, bts
8	FLOAT	addsd, addss
9	HALT	hlt
10	JUMP	jmp, ljmp
11	LEA	lea
12	LOGIC	and, not
13	MISCELLANEOUS	bound, enter
14	RETURN	ret, lret
15	SIGN	cbw, cdq
16	STACK	pop, push
17	STRING	cmpsb, stos
18	SYSTEM CALL	int, syscall
19	TEST	cmp, test

Table 1: The first column shows the position in the array assigned to each class (bit 0 is reserved for fingerprint separators in the database); the second column specifies the names of the classes that we considered; the last column provides some examples of the instructions included in each class.

might move up the preparation of the inputs or postpone the processing of the output by injecting junk instructions and control flow alterations between those two phases and the actual invocation of the API, in an attempt to evade the API detection performed by Apìcula. Moreover, it should be noted that a computational path may be shorter than the fingerprint length, since either the partial computational path that leads to the central node or the one that starts from it can be shorter than half the fingerprint length: in that case, we pad the fingerprint by adding special colors whose bits are all set to 0. In particular, if the *shorter* computational path is the one starting from *before* the block to be fingerprinted, then the padding colors will be added at the *beginning* of the fingerprint; on the other hand, if the shorter computational path is the one that starts from the block to be fingerprinted, then the padding colors will be added at the end of the fingerprint.

Given a certain length k and a list of basic blocks l, Apìcula generates the fingerprint set for a basic block $b \in l$ by first visiting all the blocks reachable from b by going *backward* up to a depth of $k/2$ ($+1$ if k is even) nodes, then by visiting all the blocks that are reachable from b by going *forward* up to a depth of $k/2$ nodes, and finally by merging all the partial fingerprints computed by first going "backward" and then going "forward", so to obtain all possible combinations of those partial fingerprints. Figure 2 shows how the set of fingerprints for a given basic block is computed. The single (partial) fingerprint is obtained by concatenating the colors of the blocks that are encountered while traversing the graph through a depth-first search that starts from the block to be fingerprinted.

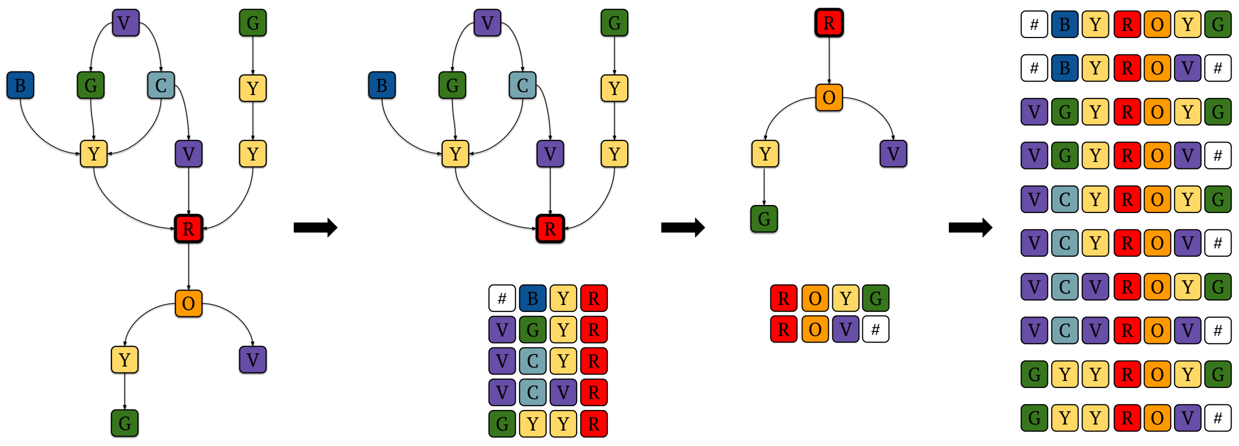

Figure 2: The fingerprinting process for $k = 7$. Each basic block in the figure is filled with a color and marked with the corresponding letter (for example, filled with yellow and marked with a Y), which symbolically correspond to the colors actually employed in Apìcula; white basic blocks with a # written inside denote padding colors. In this case, the basic block that is being fingerprinted is the red, central one.

3.4. Matching

After the fingerprint sets are generated for each basic block containing a call instruction[2], the last step in the Apìcula's work-flow is to determine whether any of those basic blocks is actually invoking one of the APIs that we are interested in detecting. As it was already mentioned before, the general idea is to compare the fingerprint set found for a single node against a database of pre-computed fingerprints: if the fingerprint set matches the database with a certain *score*, then Apìcula infer that the node is invoking that API. This matching is implemented through the evaluation of the *overlap* coefficient between the two sets (namely, the fingerprint set associated with the basic block and the database of fingerprints). We chose the overlap coefficient over other similarity measures because we are comparing small sets (a few hundred bytes of information on average) against potentially much larger ones and, therefore, it would be inappropriate to use scores that place more emphasis on the actual similarity between sets (like the Jaccard index). In fact, the overlap coefficient is not penalized in case the set of fingerprints of one API in the database is much larger than the average fingerprint set computed for blocks that invoke it, since it only measures the extent to which a smaller set overlaps with a greater one. Equation 1 shows how the overlap coefficient is computed for two sets A and B.

$$OverlapCoefficient(A, B) = \frac{|A \cap B|}{min(|A|, |B|)} \tag{1}$$

After the fingerprinting process has been completed, Apìcula iterates over the basic blocks with a call instruction, computing the overlap coefficient between their fingerprint sets and the set of fingerprints of each API in the database: if the overlap score between the fingerprint set of a block b and an API A is greater than a given threshold, then Apìcula outputs that b invokes A. Although we leave the possibility of choosing the preferred threshold, we set it to 0.55 by default. In the

[2]again, to an external address.

experiment described in Section 4.2, we observe that a more conservative threshold can be set in an *ideal* case, such as extracting the database of fingerprints from binaries that are very similar to those that are used for testing. Nonetheless, our less conservative threshold proves to be reliable for a large group of APIs when tested over a more realistic set of binaries, as shown in the experimental evaluation presented in Section 4.3. In general, it should be noted that Apìcula might output more than one API for a single block. However, we aim at eliminating this occurrence by restricting the APIs, which we wish to detect, to those that yield a low number of false positives.

3.5. Fingerprint Database Generation

In order to generate a database of fingerprints, we require a large set of executable programs containing complete information on the imported libraries within their headers. This might sound in contradiction with what we have said in the beginning, namely that Apìcula is meant to work on generic byte streams and does not require access to the header information of an executable file: in reality, we are not contradicting our original claim at all, since this is only required for building the database that will then be used to identify APIs in truly generic streams of bytes.

For this purpose, we focus on Windows Portable Executables (PEs) [27], mainly because it is relatively easy to access a wide number of malware samples for the Windows operating system. In short, a PE contains all the information needed by the Windows operating system to load the executable code in memory and manage its execution. To do this, it is composed of several headers and sections. However, for our goal, we only focus on the *import directory table* and the *.text* sections.

The *import directory table* contains information on the Dynamic-link Libraries (DLLs) that are imported by the executable. In particular, each entry in the table carries the address information that is used to resolve the entry points in the DLL, most notably the Import Lookup Table (ILT) and the Import Address Table (IAT). An ILT holds the names (or ordinals) of the objects imported from a DLL, and we use them to identify which API functions are imported within the executable. On the other hand, an IAT is a table of pointers where each entry contains the address of a symbol imported from a DLL: we use these tables to reconstruct where an API call is performed in the code.

On the other hand, the *.text* section contains the executable code. In Windows, x86 executable code might contain a *jump stub* for each API function used within the program. These jump stubs constitute an additional layer of indirection inserted by the compiler to simplify the task of overwriting the relative virtual addresses with the effective addresses at which the libraries are loaded at run time. These stubs are made of one jump instruction to the corresponding IAT entry: when the code needs to invoke a certain library function, it *calls* the corresponding jump stub instead, which will then jump to the actual address specified in the IAT. Alternatively, the function pointers in the IAT might be invoked directly. Figure 3 shows examples of a direct call to an IAT entry and an indirect call through a jump stub.

3.5.1. Database Generation Process

Schematically, Apìcula builds the fingerprint database by iterating over a list of portable executable files: for each of them, it reads the ILTs and the IATs for all the imported DLLs, constructs a table in which the name of every API is mapped to the address of the IAT entry that stores its address, and then extracts the actual executable code along with its static address, which is computed as shown in Equation 2.

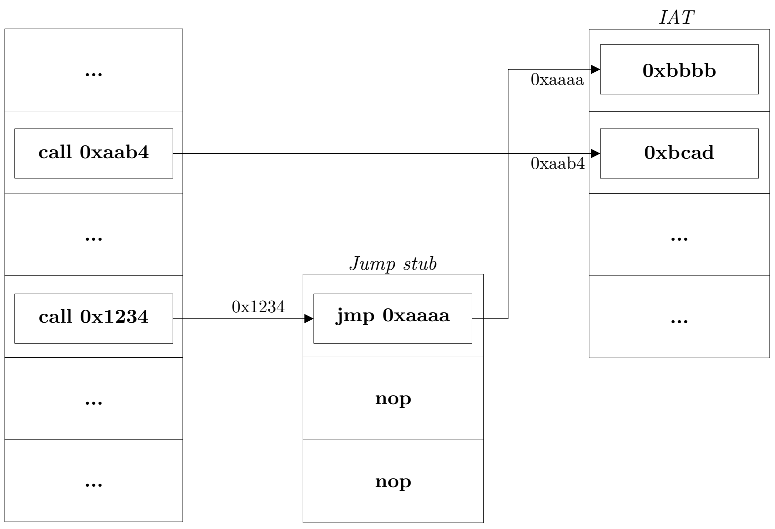

Figure 3: Examples of direct a call to a IAT entry and of an indirect call through a jump stub.

$$Address = ImageBase + CodeVirtualAddress \tag{2}$$

These initial steps are attained by reading and parsing the information contained within the header of the PE file. Since we wrote Apìcula in Python, we implemented this through the *pefile* library [28].

After this preliminary phase is complete, Apìcula continues with the construction of the CFG, as described in Section 3.2: the only exception, in this case, is that, during the disassembling of the code, the jump stubs are identified and saved in an appropriate data structure that keeps track of the correspondence between APIs and jump stubs. Next, blocks that contain a call instruction to an API are fingerprinted, and the resulting fingerprint sets are appended to the corresponding database entry, which is essentially a regular file named as the API function it refers to (e.g., *kernel32-createthread*). This is attained by identifying the correct API invoked by each block, through a look-up to the information held by the data structures described above (namely, the jump stubs table and the data structure that maps the names of the APIs to their corresponding IAT entries). Finally, in a second separate phase, all the files are cleaned by merging together single fingerprint sets while removing duplicated fingerprints. We decided to keep these two phases separated because this would allow us to easily shorten the fingerprint length without having to re-run the entire database construction process.

4. Validating the Effectiveness of Apìcula

In this section, we describe the experimental evaluation conducted to prove the effectiveness of Apìcula. In the first experiment, we verify that Apìcula can distinguish among different APIs by measuring the similarity between sets of fingerprints belonging to blocks containing a call instruction to an API function. In the second experiment, in favor of this work's reproducibility, we present the approach that we took for tuning the fingerprint length and the overlap threshold to maximize the accuracy of Apìcula. Lastly, the third experiment identifies the specific APIs that we can match with a certain degree of confidence across a more significant and heterogeneous dataset composed of malicious samples. In all experiments, we work under the Windows environment, as the long-term goal of our research is to simplify the behavioral analysis of malware.

4.1. Experiment 1: Fingerprint Similarity

With this experiment, we aim at demonstrating that the fingerprint sets generated from basic blocks that invoke different APIs are statistically dissimilar among them. In other words, we want to prove that the fingerprint sets generated by Apìcula are unique for each API, as this would otherwise produce a high misclassification rate. We show this by first averaging the similarity scores among basic blocks that invoke the *same* API and, then, by comparing those results against the average scores computed for basic blocks that invoke *different* ones. We repeat this across four optimization levels, namely $O1$, $O2$, $O3$, and Os.

The dataset we perform this experiment on is the GNU core utils, a collection of the "basic file, shell and text manipulation utilities of the GNU operating system" [29]. We obtain the source code from the official Github page [30] and build them for Windows through the *MinGW* [31] port of GCC within a *Cygwin* environment [32].

The experiment is composed of two separate parts: we first build our binaries with the same optimization level and show how the fingerprint sets associated with the *same* API are statistically similar, while those associated with *different* APIs are *remarkably dissimilar* among them; secondly, we show how compiling programs with *different* optimization levels causes the average similarity scores - among blocks with the same API - to deteriorate significantly. However, it is important to point out that, even when considering binaries compiled with different optimization levels, we can still tell apart blocks that contain an invocation to the same API from blocks that contain a call to different ones. Nonetheless, the deterioration of the average similarity values between fingerprint sets associated with the same API is important to explicitly point out the necessity to build databases of fingerprints that are sufficiently diversified to *capture* all the possible variations when using Apìcula on real-world binaries.

For both the sub-experiments, we employ the Jaccard index to compare the similarity among the fingerprint sets of basic blocks. The Jaccard index (which is described in Equation 3) is preferred over the overlap coefficient, used for matching fingerprint sets against the API database in Apìcula, because it measures the overall similarity between two sets with similar size and not just the extent to which a smaller set overlaps with a bigger one. As a matter of fact, in the matching phase of Apìcula, fingerprint sets are compared against a database of fingerprints whose size is significantly greater in most cases, while, in this experiment, the size of the fingerprint sets across different blocks belongs to the same order of magnitude.

$$Jaccard(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{3}$$

4.1.1. Experiment 1.1: Fingerprint Similarity with same Optimization Level

In this experiment, we show that the average similarity among fingerprint sets associated with different APIs is significantly lower than the average similarity of fingerprint sets associated with

	O1	O2	O3	Os
Same API (α)	0.30	0.28	0.26	0.32
Different APIs (β)	0.0002	0.0001	0.0001	0.0001

Table 2: Average similarity scores between fingerprint sets associated with the same API (α) as well as the average similarity scores of fingerprint sets associated with different APIs (β), sorted by optimization option.

the same API. In particular, we perform the following computations for all the four considered optimization levels separately.

First, we compute the average similarity between blocks with the same API call. We do this on an API basis, meaning that we repeat this procedure for all the APIs that have been encountered within the dataset. In particular, we compute and then average the similarity scores between all the possible pairs of fingerprint sets that correspond to the same API. Equation 4 shows how this computation is performed (for a given optimization level):

$$\alpha = \frac{1}{n}\left[\sum_{A \in APIs}\left(\sum_{(I,J)\in\theta(A)} Jaccard(I,J)\right)\right] \tag{4}$$

In Equation 4, θ is a function that outputs the superset of all the possible pairs of fingerprint sets (I, J) observed for a given API A, whereas n is the overall number of these pairs across all $A \in APIs$. Since the Jaccard index is commutative, pairs (A, B) and (B, A) are considered equivalent and included in the superset outputted by θ only once.

Next, we compute the average similarity score between blocks that invoke different APIs. Given two APIs A_1 and A_2, the similarity score is obtained by comparing all the fingerprint sets belonging to A_1 against all the fingerprint sets belonging to A_2, and then averaging the results; this is repeated for all the possible combinations of different APIs, namely for all the pairs (A_i, A_j) with $A_i \neq A_j$, as shown by Equation 5.

$$\beta = \frac{1}{m}\left[\sum_{(A_i,A_j)\in\gamma} \frac{1}{q}\left(\sum_{I\in\delta(A_i)}\sum_{J\in\delta(A_j)} Jaccard(I,J)\right)\right] \tag{5}$$

In Equation 5, γ is a function that outputs all the possible pairs of distinguished APIs (A_i, A_j), $A_i \neq A_j$, whereas m is the cardinality of γ's codomain; lastly, δ returns the superset of all the fingerprint sets observed for a given API, and $q = |\delta(A_i)| * |\delta(A_j)|$.

Table 2 shows the average similarity scores between fingerprint sets associated with the same API, as well as the average similarity scores of fingerprint sets associated with different APIs across the four optimization options that we have considered. As the average value of α (i.e., same API) is 0.29 and the average value of β (i.e., different APIs) is 0.000125, we can conclude that (1) there is a remarkable difference in the order of magnitude between the two measures, and (2) the similarity of basic blocks that contain a call instruction to different APIs is notably low. In light of these aspects, we can claim that Apìcula can distinguish between different APIs with strong statistical confidence.

4.1.2. Experiment 1.2: Fingerprint Similarity with Different Optimization Level

In the previous experiment, we show that Apìcula can distinguish between the invocation of different APIs by comparing the average similarity scores of the corresponding fingerprint sets when compiling the binaries with the same optimization level. In this experiment, we want to verify whether Apìcula can distinguish between fingerprint sets associated with different APIs when these are extracted from binaries compiled with different optimization levels.

To this end, for each API, we compute the Jaccard index between the fingerprint sets extracted from programs built with the $O1$ optimization against the fingerprint sets extracted from programs compiled with either $O2$, $O3$, or Os. Next, we average these values for each pair of optimization options (i.e., $O1\&O2$, $O1\&O3$, and $O1\&Os$). The resulting values are then compared against the average score obtained for fingerprint sets associated with different APIs, as in the previous experiment. However, we do not compute the average similarity between fingerprint sets associated with different APIs when these are extracted from binaries compiled with *different* optimization levels. On the contrary, we directly consider the average of the values that were obtained in the previous experiment (i.e., 0.000125). We do this since the average similarity across programs compiled with the same optimization option is generally greater than the similarity expected from programs compiled with different optimization options. In other words, by doing so, we are actually performing a stricter comparison.

As shown in Table 3, a significant loss of similarity can be observed with respect to the previous experiment: the average value of α was, in fact, 0.29, whereas the average value we obtained with this experiment is instead 0.07, that is to say, the average value decreases by 76%. Notwithstanding this pronounced loss in the average similarity, we can still confidently distinguish between fingerprints associated with different APIs, as the average value of β remains significantly smaller.

4.1.3. Additional Discussion on Results

It is important to stress once again that the goal of experiment 1 is to demonstrate that Apìcula can distinguish among different APIs. It shows that the fingerprint sets generated from basic blocks that invoke different APIs are statistically dissimilar. In the process of proving this, we also compute the similarity among fingerprint sets associated with the *same* API. Particularly in experiment 1.1, we observe a significantly high similarity score between fingerprint sets associated with the same API. Nonetheless, we do not expect the similarity score to be as consistently high when comparing fingerprint sets extracted from binaries belonging to more heterogeneous datasets, even though they refer to the same API function. As a matter of fact, we believe that the *average* similarity scores are particularly high because of the inherent characteristics of the dataset used, such as the limited size and the fact that the programs are mainly authored by a restricted group of people (i.e., we expect code re-use to play a significant role in this). Furthermore, by mixing optimization options, we observed a significant decrease in the average similarity (76%). However, a lower similarity (on average) between fingerprint sets associated with the same API does not constitute a limitation of Apìcula, since the detection of API calls is ultimately achieved by constructing a sufficiently sizable database of fingerprints (the following experiments will deal with this).

O1&O2	O1&O3	O1&Os	Average β
0.08	0.08	0.05	0.000125

Table 3: Similarity scores across different optimization levels: the values are obtained by averaging the Jaccard index computed between basic blocks with the same API call, that were taken from programs compiled with the option $O1$ and then $O2$ (first column), with $O1$ and then $O3$ (second column), and finally with $O1$ and then Os (third column). The last column contains the comparison value that we use to determine whether Apìcula can distinguish between different APIs.

Moreover, throughout this first experiment, we always consider average values and never concentrate on the underlying statistical distribution: namely, we do not disclose any specific information on the APIs observed within the dataset. This has been done voluntarily, as the goal of this first experiment is exclusively to show that Apìcula can statistically tell apart different APIs. That being said, we remind the reader that our original claim was actually that only *some* APIs possess fingerprints that are shared across different binaries and at the same time sufficiently unique. In Section 4.3, we will address the problem of identifying which APIs possess fingerprints with the said qualities.

4.2. Experiment 2: Average Overlap Values and Optimal Fingerprint Length

The goal of this second experiment is to verify whether, provided a database of fingerprints, Apìcula is actually capable of identifying API calls in binaries. We still maintain the same experimental setting that we used in Section 4.1; namely, we perform the experiment over the GNU core utils. In particular, we build the programs in that collection using all the four different optimization options that we have considered so far; we then shuffle the resulting binaries and randomly select 70% of them, which we use to build the database of fingerprints. Instead, the remaining 30% is used to test whether Apìcula can effectively employ that database to identify API calls. To this end, we measure the overlap coefficient between the fingerprint set of each basic block b and the database of fingerprints of the corresponding API that is invoked by b (which we can derive from looking at the information stored in the headers of the corresponding PE file), averaging the results. The following Equation (6) shows how this is attained formally:

$$\phi = \frac{1}{n}\left(\sum_{i=0}^{n} OverlapCoefficient(b_i, f(b_i))\right), \tag{6}$$

where b_i is the i^{th} basic block with a call instruction, n is the total number of such basic blocks and f a function that outputs the set of fingerprints in the database associated with the API invoked by b_i.

Similarly, we compute the overlap coefficient between each basic block b and the fingerprint set of each API in the database, excluding the one that was actually invoked by b; we then average the results (Equation 7).

$$\psi = \frac{1}{m}\left(\sum_{i=0}^{n}\sum_{A_j}^{g(b_i)} OverlapCoefficient(b_i, A_j)\right). \tag{7}$$

In Equation 7, b_i is the i^{th} basic block containing a call instruction, whereas g is a function that returns the superset of all the fingerprint sets in the database, excluding however the fingerprint set of the actual API invoked by b_i; lastly $m = n * (|APIs| - 1)$, where n is the number of basic blocks with a call instruction (as in Equation 6), and $|APIs|$ the overall number of APIs in the database. The values that we computed for ϕ and ψ allowed us to claim that, in this scenario, we can identify the correct API calls without producing any false positive by using a default threshold of 0.55.

We take advantage of the relatively small size of the dataset and repeat the experiment for different fingerprint lengths in an attempt to identify an optimal value or range that could be used for real-life datasets as well. In particular, we run the experiments starting from a considered length of 9, and then go down all the way to 4. We measure the performance for a given fingerprint

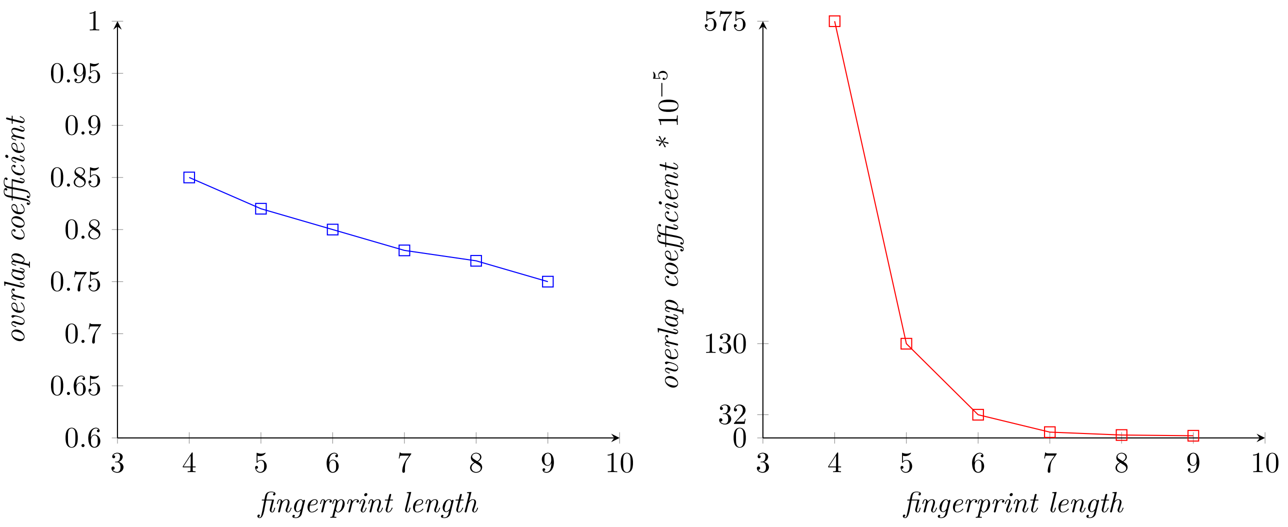

Figure 4: The graph on the left shows the values of ϕ for fingerprint lengths in $[4, 9]$, whereas the graph on the right shows the values of ψ in the same interval.

length k by considering both the average overlap coefficient computed for "matching" fingerprint sets and APIs (ϕ_k), and the one computed between "non-matching" fingerprint sets and APIs (ψ_k). All the six considered lengths performed relatively well in expressing the similarity between blocks with the same API call. In particular, 4 was the one that performed the best: this was expected since shorter fingerprints are more generic and therefore can be matched more easily. Nonetheless, for the very same reason, shorter and thus more generic fingerprints also imply an increment in the average similarity between "non-matching" fingerprints (ψ). In particular, an increment by an exponential factor in the value of ψ can be seen when shortening the fingerprint length from 7 to 4. We ultimately chose 6 as a reference value, since it constitutes a fair compromise between the resulting values of ϕ and ψ, and the expected computational performance, given that a length of 6 allows us to have smaller databases and, therefore, to compute the overlap coefficient more efficiently[3].

4.3. Experimenting with Real Malware

In the previous experiments, we show that the coloring system of Apìcula can yield fingerprint sets that are similar across different programs and that we can build a database of fingerprints which Apìcula can employ to identify APIs in binaries. Both experiments are carried out on a specific dataset (namely, the GNU core utils), which in our opinion represents an "ideal" scenario for Apìcula, due to the intrinsic similarity that is expected throughout the different programs in the dataset. Moreover, the (relatively) small size of the dataset allows us to repeat the second experiment several times, so to estimate an ideal fingerprint length that can be used on more realistic datasets as well. In both the experiments, we do not pay particular attention to which specific APIs we are matching. On the contrary, we arbitrarily consider average values, which turn out to be significantly high for "matching" APIs, and remarkably low for "non-matching" ones: in other words, the results allows us to conclude that - when used on that *specific* dataset - Apìcula

[3]when compared to greater fingerprint lengths.

can truly identify APIs with outstanding statistical confidence. Unfortunately, we do *not* expect the same to happen for larger, more diverse, as well as more realistic datasets: in truth, in the average case, we do not even expect to be able to match all APIs. In fact, our original idea is based on the intuition that *some* APIs require specific input preparation and return some output which is potentially processed following the API invocation itself; hence, Apìcula is designed to identify APIs by fingerprinting the code that *surrounds* a call instruction. However, not all APIs require the preparation of special inputs nor return an output that needs processing.

The goal of this third experiment is twofold: first, we want to show that Apìcula can be used to identify specific APIs in real-life malware datasets; then, we want to actually find a subset of the WinAPIs that we can identify with good confidence and without incurring in too many false positives. For this last purpose, we first introduce and define two essential concepts.

Definition 1: ***API fingerprintability.*** By API fingerprintability, we mean the ability to correctly identify an API through a database of its fingerprints. In order for an API to be "fingerprintable," we expect the number of *true positives* (TPs) to be significantly greater than the number of *false negatives* (FNs). For this reason, we consider an API to be fingerprintable if its observed *recall* (Equation 8) is greater or equal than a certain threshold: in our case, we considered a threshold of 0.8.

$$Recall = \frac{TPs}{TPs + FNs} \tag{8}$$

Definition 2: ***API fingerprint uniqueness.*** By API fingerprint uniqueness, we mean that the fingerprints collected for a specific API are sufficiently unique not to be confused with those of other APIs. Since it is unfeasible to compare the APIs we are interested in against *all* the possible APIs in the world, we only expect APIs to be sufficiently unique among the ones that we are actually fingerprinting. For an API to be considered *sufficiently* "unique," we expect the observed false positives (FPs) to be adequately low, especially in relation to the number of observed TPs. For this reason, we consider the fingerprint set of an API to be sufficiently unique if the observed *false discovery rate* (Equation 9) is smaller than a certain threshold, in our case 0.05.

$$FDR = \frac{FPs}{FPs + TPs} \tag{9}$$

To carry actual statistical relevance, both properties need to be confirmed over a large number of samples since they are evaluated through data analysis metrics (i.e., recall and false discovery rate). In this experiment, we consider a dataset that comprises 6000 unpacked malware samples retrieved from Virusshare [33]: in particular, we select a malware bundle of 130000 samples and randomly extract 6000 from it. We use half of those (i.e., 3000) to build the database of fingerprints and the other half to identify APIs that possess both the properties described above. For this test, we use a fingerprint length of 6 (for the reasons highlighted in the previous experiment) and an overlap threshold of 0.55. We identified a total number of 1510 different APIs belonging to 34 different DLLs. Of these, 427 APIs were *fingerprintable*, while 271 had *unique* fingerprints: finally, a total number of 228 APIs were found to be both fingerprintable and to have unique fingerprints. These all belong to either *user32.dll* (63), *kernel32.dll* (142) or *msvcrt.dll* (23).

To provide further details on the APIs that we found to possess both properties, we group them in the ten classes that are reported in Figure 5. It must be noticed that an API could be assigned to more than one class; for example, many APIs belonging to the *file* class could be assigned to the *input/output* class as well. In those corner cases, we assign the APIs to the classes that best describes them.

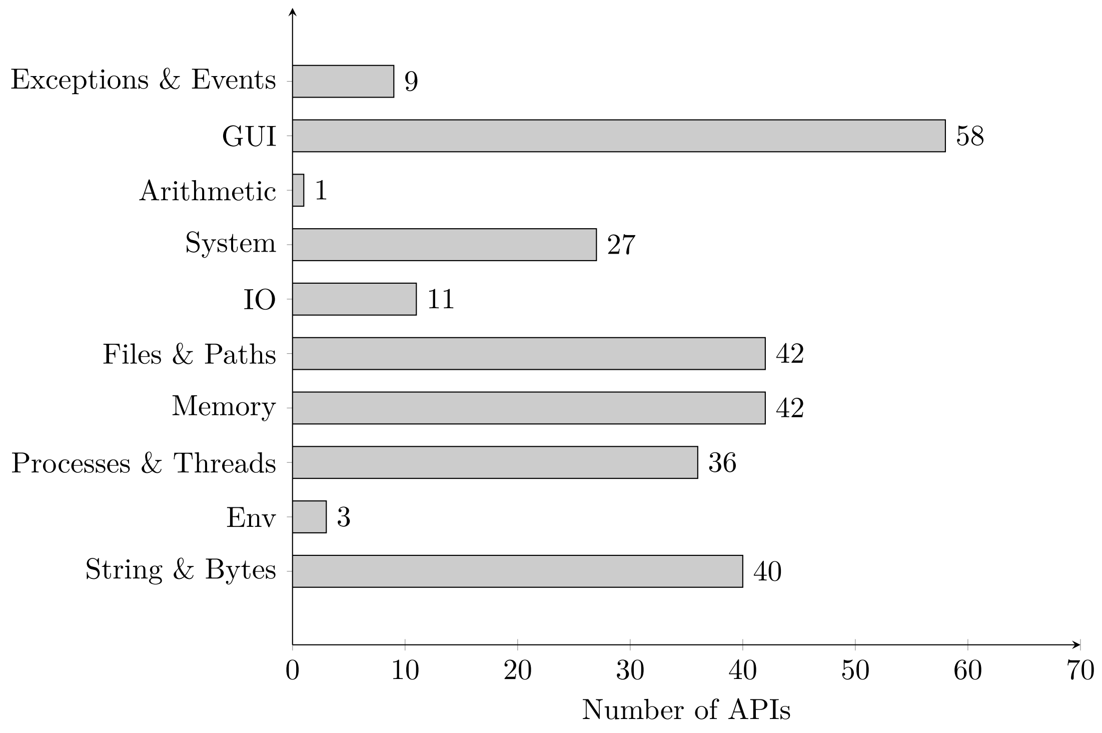

Figure 5: The histogram shows the ten classes in which we grouped the APIs that we found both to be fingerprintable and to possess sufficiently unique fingerprints.

The following are examples of the APIs that are identifiable by Apìcula:

- *kernel32.dll-isdebuggerpresent* and *kernel32.dll-outputdebugstringa*, employed by evasive malware to check whether they are being executed within a debugger [34, 35] (we assigned it to the *system queries* class);
- many file manipulation APIs such as *kernel32.dll-createfile*, *kernel32.dll-writefile* and *kernel32.dll-deletefile*, as well as some file discovery APIs, for example *kernel32.dll-findfirstfilea* and *kernel32.dll-findnextfilea*;
- *kernel32.dll-loadlibraryw*, which loads a library in the memory of a process;
- several APIs for creating and manipulating processes and threads, for example *kernel32.dll-createprocessa*, *kernel32.dll-createthread* and *kernel32.dll-openprocess*;
- many memory management APIs, such as *kernel32.dll-virtualalloc*, *kernel32.dll-virtualfree*, *kernel32.dll-virtualquery*, and *kernel32.dll-globallock.*

5. Validating the Fingerprint Database

In Section 4, we demonstrate that Apìcula is able to identify APIs in binaries; more specifically, in Section 4.3 we build a database of fingerprints from a dataset of real malware and test it against a separate set of malicious programs. This allows us to show that Apìcula can work on real malware datasets, and that it can identify specific APIs with high probability. However, the approach we take for the latter is computationally expensive and has the downside of requiring separate datasets to construct the database and measure the performance of each API. Given that someone might want to build a new database of fingerprints (for example, in order to include some specific software library), we want to provide a method for validating the quality of the database, in particular for identifying the APIs that are fingerprintable and have sufficiently unique fingerprints without having to recur to the more complex approach described in Section 4.3. To this end, we take advantage of the results obtained from the experiment in Section 4.3 to derive two strategies that can allow a user to *infer* which APIs possess the two properties mentioned above.

5.1. Determining Fingerprintability from Database Size

In order for Apìcula to correctly identify the API A invoked by a basic block b, its database entry for A must contain at least some of the fingerprints in the fingerprint set of b: this means that, to some extent, some of the fingerprints in the fingerprint set of b must have already been observed in some other binary. More precisely, we expect Apìcula to *encounter* the same fingerprints several times and across multiple binaries while constructing the database of fingerprints. Clearly, this is not always true, as utilizing a low number of samples for constructing the database of fingerprints might preclude the observation of *repeated* fingerprints altogether. Nonetheless, when an appropriately sized dataset (such as the one that we employed) is used to construct the database, we can confidently expect to observe repeated fingerprints for a given API (if this is truly fingerprintable, that is to say).

We exploit this intuition and claim that, in the average case, we can identify a fingerprintable API by considering how much the byte size of the fingerprint database associated with it *shrinks down*, when this is processed during the second phase of the database construction, namely when the observed sets of fingerprints are merged into a single set, and all duplicated fingerprints removed. In particular, we are looking for a real number $p \in (0, 1]$, such that the following inequality holds true:

$$|DB_A^{clean}| \leq p|DB_A^{raw}| \tag{10}$$

where $|DB_A^{clean}|$ is the size of the fingerprint database for an API A after the merging and removal of duplicated fingerprints, while $|DB_A^{raw}|$ is the size of the raw database obtained after the first part of the database construction.

In order to find an appropriate value for p, such that an API could be safely deemed fingerprintable, we estimate how much the fingerprint sets for the APIs that we found to be fingerprintable in Section 4.3 shrink down on average; we then use the resulting value as an upper-bound for p. Furthermore, to show that our claim is sound, in particular, that this value of p can indeed be used to tell apart fingerprintable APIs from *non*-fingerprintable ones, we also perform the same computation for the APIs that were not identified as fingerprintable in Section 4.3. Figure 6 shows the results that we obtained; more specifically, it plots the average values of p for the APIs with observed recall in the corresponding recall range (the recall ranges that we considered are $[0.0, 0.05)$, $[0.05, 0.10)$, $[0.10, 0.15)$, and so on). Setting p to an upper bound of 0.266 allows us to label all fingerprintable APIs with an accuracy of 99.6%, while generating only 6 false positives out of the total 1510 APIs that were identified in Section 4.3. This upper-bound can therefore be used to identify fingerprintable APIs when constructing new fingerprint databases.

5.2. Fingerprint Uniqueness

In Section 5.1, we propose a method that can potentially allow us to determine which API is fingerprintable, without having to recur to the computationally demanding approach taken in

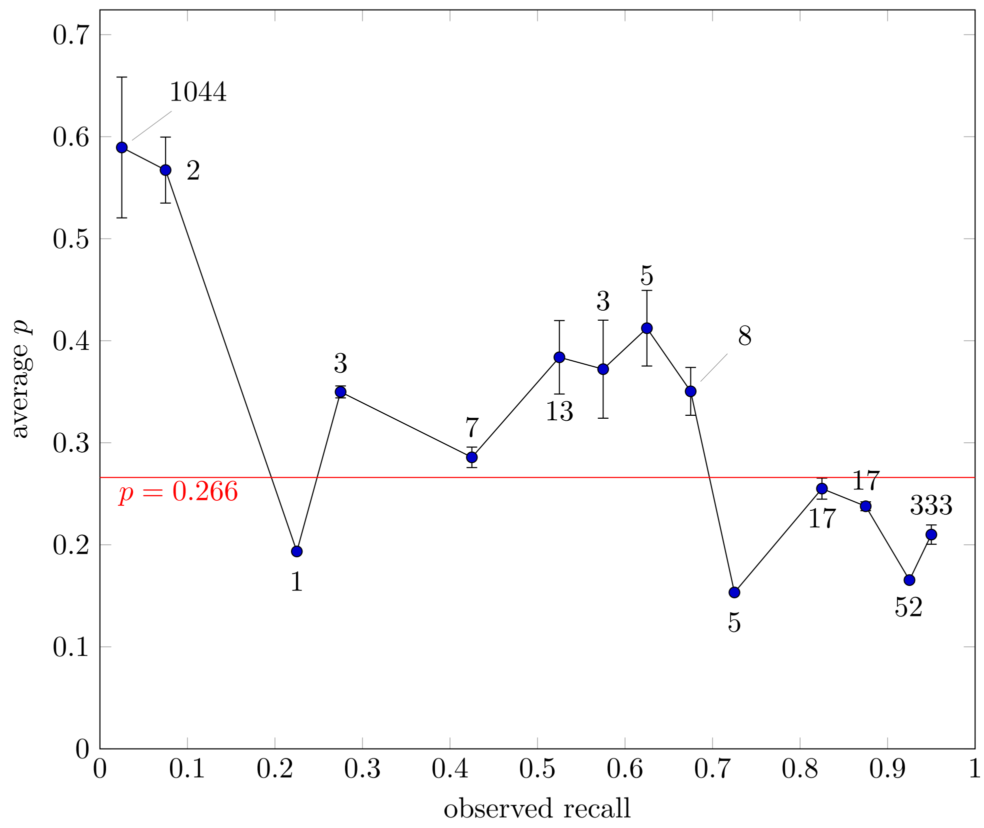

Figure 6: the graph shows the average values of p for APIs with observed recall in the corresponding recall range (i.e., $[0.0, 0.05)$, $[0.05, 0.10)$, $[0.10, 0.15)$, and so on). Each point is labeled with the number of APIs found in that range. The red horizontal line indicates the upper-bound that we select for p.

Section 4.3. In this subsection, we complete our discussion on the validation of a database of fingerprints by introducing an alternative method for identifying which APIs possess *sufficiently unique* fingerprints, according to the property first defined in Section 4.3. To this end, we evaluate the overlap coefficient (described in Equation 1) between an API A_i and all other APIs in our database: if *all* the values are smaller than a certain upper-bound q, then we claim that A_i possesses fingerprints that are sufficiently unique among the APIs that we wish to identify with Apìcula. Conversely, an overlap value computed between A_i and another API A_j, which is greater than the estimated q, might indicate that both APIs might yield at least some FPs and, therefore, should be ideally eliminated from the database altogether.

Similarly to what we do for the first property, we exploit the results obtained in Section 4.3 to estimate an ideal value for q. Consequently, we expect this method to properly function only when the database is constructed from a sufficiently large number of executable programs (for instance, greater or equal to the one that we employ, i.e., 3000 samples). To find an optimal value for q, we divide all the APIs found in Section 4.3 into 20 distinct groups, based on their observed false discovery rate: in particular, we consider 20 distinct ranges (i.e., $[0.0, 0.05)$, $[0.05, 0.10)$, $[0.10, 0.15)$, and so on).

We compute the overlap coefficient for each pair of fingerprint sets associated with two distinguished APIs (A_i, A_j); we then average the results first on an API basis and then on a range basis, that is to say, for all the APIs with false discovery rate in the range $[r_a, r_b)$, we first average all the overlap scores computed for each of those APIs. Then, we average all the resulting mean values in the range $[r_a, r_b)$. The reader should notice that, given two APIs A_i and A_j, if the respective false discovery rates belong to two *different* ranges, then the resulting overlap scores will be included separately in the average value computed for both ranges. Figure 7 shows the average overlap scores that we obtain for each considered range. An upper-bound $q = 0.0005$ can be set to distinguish APIs with observed false discovery rate smaller or equal to 0.05, with an overall accuracy of 100%.

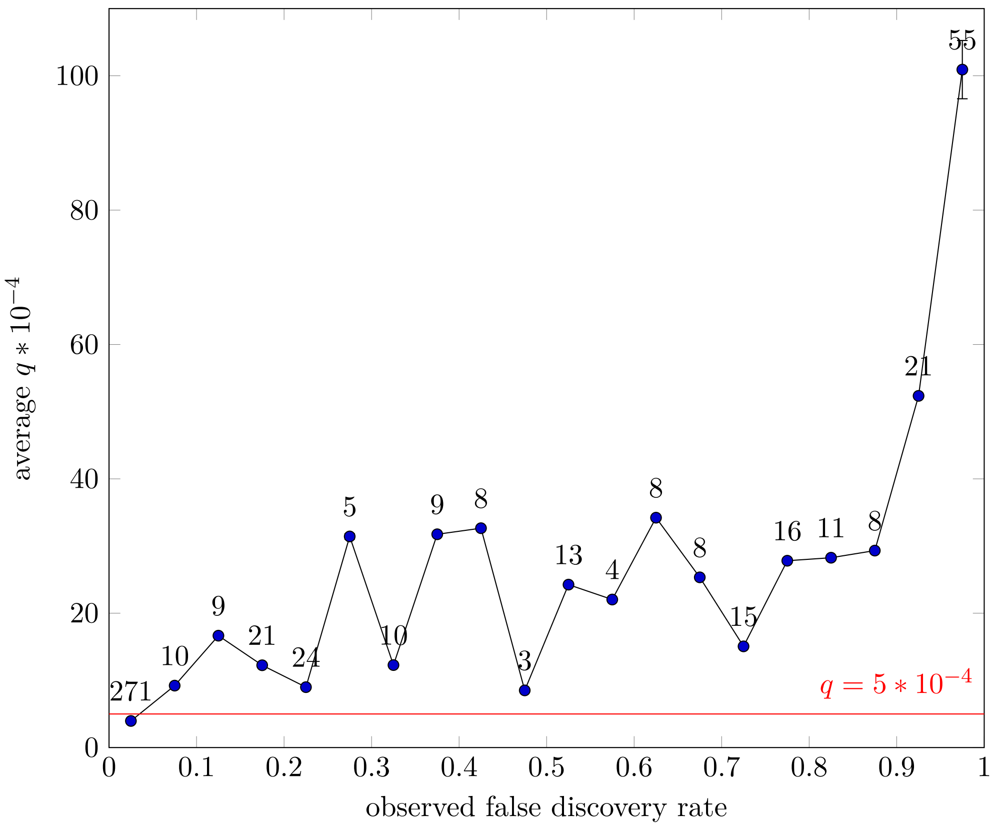

Figure 7: the graph shows the average values of q for APIs with observed false discovery rate in the corresponding recall range (i.e., $[0.0, 0.05)$, $[0.05, 0.10)$, $[0.10, 0.15)$, and so on). Each point is labeled with the number of APIs found in that range. The red horizontal line indicates the upper-bound that we select for q.

6. Conclusions and Future Work

In this work, we show how the fingerprinting of code can be used to identify APIs in unseen binaries. In particular, we focus on the fingerprinting of the code that precedes and immediately follows a call instruction, as those two sections usually contain the code responsible for preparing the inputs to an API function and for processing its output (if any). Through the experiments that we conduct, we manage to validate our original claim, that is to say, for some APIs, the code responsible for the preparation of the inputs and the processing of the output is shared across different binaries and sufficiently unique to be fingerprinted. Moreover, we test our tool against a set of real-life malware samples, which allows us to show how Apìcula can indeed be used as a complementary tool for analyzing malicious binaries. Furthermore, we isolate a subset of the Windows APIs that can be detected with sufficient sensitivity and negligible false discovery rate.

Notwithstanding the efficacy shown throughout the experiments that we perform, our approach suffers from most of the limitations that are associated with other static analysis techniques, such as the scarce resilience to obfuscation and packing. In particular, obfuscation techniques, such as the injection of junk instructions, could be used to generate basic blocks whose colors have most bits set to 1, which would result in very poor performances by Apìcula. Similarly, the code responsible for the preparation of the inputs to an API might be anticipated in the flow of the program, while the code responsible for the processing of the output could be postponed in order to avoid being fingerprinted by Apìcula: in these cases, we could extend the fingerprint length, although that would worsen the time performance of Apìcula, as longer fingerprints would imply an additional overhead. On the other hand, although a packed binary would not be directly analyzable by Apìcula, we do not worry excessively about packing and encryption because we imagine Apìcula being primarily employed on memory dumps, which can be expected to contain at least some parts of the binary that are unpacked or decrypted.

In conclusion, we believe that Apìcula could be used as a base for more advanced tools. In particular, we intend to employ Apìcula to perform automated malware classification, similarly to what has already been done in [1, 2, 3] (dynamic analysis) and in [11] (static analysis). However, in our case, we aim at automatically classifying byte streams that can neither be executed nor carry headers information that can be exploited to reconstruct traces of API calls, for instance, incomplete memory dumps or files containing only object code. Moreover, we intend to experiment with live memory analysis, with the goal of performing live API call detection by directly analyzing the physical memory (for example, through a field-programmable gate array directly connected to the RAM of a computer). Lastly, we intend to integrate Apìcula in Jackdaw, a hybrid-analysis tool meant to produce human-readable reports to aid analysts in performing manual malware analysis [14].

Acknowledgment

Mario D'Onghia acknowledges support from TIM S.p.A. through the PhD scholarship.

References

[1] K. Rieck, P. Trinius, C. Willems, T. Holz, Automatic analysis of malware behavior using machine learning, Journal of Computer Security 19 (2011) 639–668. doi:10.3233/JCS-2010-0410.

[2] G. E. Dahl, J. W. Stokes, L. Deng, D. Yu, Large-scale malware classification using random projections and neural networks, in: 2013 IEEE International Conference on Acoustics, Speech and Signal Processing, 2013, pp. 3422–3426. doi:10.1109/ICASSP.2013.6638293.

[3] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, A. Hamze, Malware detection based on mining api calls, in: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 1020–1025. doi:10.1145/1774088.1774303.
URL https://doi.org/10.1145/1774088.1774303

[4] N. Peiravian, X. Zhu, Machine learning for android malware detection using permission and api calls, in: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, 2013, pp. 300–305. doi:10.1109/ICTAI.2013.53.

[5] Z. Salehi, A. Sami, M. Ghiasi, Using feature generation from api calls for malware detection, Computer Fraud & Security 2014 (9) (2014) 9–18. doi:https://doi.org/10.1016/S1361-3723(14)70531-7.
URL https://www.sciencedirect.com/science/article/pii/S1361372314705317

[6] C. H. Malin, E. Casey, J. M. Aquilina, Malware forensics: investigating and analyzing malicious code, Syngress, 2008.

[7] J. Li, D. Gu, Y. Luo, Android malware forensics: Reconstruction of malicious events, in: 2012 32nd International Conference on Distributed Computing Systems Workshops, 2012, pp. 552–558. doi:10.1109/ICDCSW.2012.33.

[8] T. Ball, The concept of dynamic analysis, in: Software Engineering—ESEC/FSE'99, Springer, 1999, pp. 216–234.

[9] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, R. Koschke, A systematic survey of program comprehension through dynamic analysis, IEEE Transactions on Software Engineering 35 (5) (2009) 684–702. doi:10.1109/TSE.2009.28.

[10] A. Moser, C. Kruegel, E. Kirda, Limits of static analysis for malware detection, in: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), 2007, pp. 421–430. doi:10.1109/ACSAC.2007.21.

[11] J.-Y. Xu, A. Sung, P. Chavez, S. Mukkamala, Polymorphic malicious executable scanner by api sequence analysis, in: Fourth International Conference on Hybrid Intelligent Systems (HIS'04), 2004, pp. 378–383. doi:10.1109/ICHIS.2004.75.

[12] S. H. H. Ding, B. C. M. Fung, P. Charland, Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization, in: 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 472–489. doi:10.1109/SP.2019.00003.

[13] S. H. Ding, B. C. Fung, P. Charland, Kam1n0: Mapreduce-based assembly clone search for reverse engineering, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 461–470. doi:10.1145/2939672.2939719.
URL https://doi.org/10.1145/2939672.2939719

[14] M. Polino, A. Scorti, F. Maggi, S. Zanero, Jackdaw: Towards automatic reverse engineering of large datasets of binaries, in: International conference on detection of intrusions and malware, and vulnerability assessment, Springer, 2015, pp. 121–143.

[15] M. R. Farhadi, B. C. Fung, P. Charland, M. Debbabi, Binclone: Detecting code clones in malware, in: 2014 Eighth International Conference on Software Security and Reliability (SERE), 2014, pp. 78–87. doi:10.1109/SERE.2014.21.

[16] W. M. Khoo, Decompilation as search, 2013.

[17] M. R. Farhadi, B. C. Fung, Y. B. Fung, P. Charland, S. Preda, M. Debbabi, Scalable code clone search for malware analysis, Digit. Investig. 15 (C) (2015) 46–60. doi:10.1016/j.diin.2015.06.001.
URL https://doi.org/10.1016/j.diin.2015.06.001

[18] Y. David, E. Yahav, Tracelet-based code search in executables, SIGPLAN Not. 49 (6) (2014) 349–360. doi:10.1145/2666356.2594343.
URL https://doi.org/10.1145/2666356.2594343

[19] W. M. Khoo, A. Mycroft, R. Anderson, Rendezvous: A search engine for binary code, in: 2013 10th Working Conference on Mining Software Repositories (MSR), 2013, pp. 329–338. doi:10.1109/MSR.2013.6624046.

[20] T. Dullien, R. Rolles, Graph-based comparison of executable objects (english version), SSTIC 5 (01 2005).

[21] M. Bourquin, A. King, E. Robbins, Binslayer: Accurate comparison of binary executables, in: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, PPREW '13, Association for Computing Machinery, New York, NY, USA, 2013. doi:10.1145/2430553.2430557.
URL https://doi.org/10.1145/2430553.2430557

[22] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, G. Vigna, Polymorphic worm detection using structural information of executables, in: Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection, RAID'05, Springer-Verlag, Berlin, Heidelberg, 2005, p. 207–226. doi:10.1007/11663812_11.
URL https://doi.org/10.1007/$11663812_11

[23] P. De Nicolao, M. Pogliani, M. Polino, M. Carminati, D. Quarta, S. Zanero, Elisa: Eliciting isa of raw binaries for fine-grained code and data separation, in: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, 2018, pp. 351–371.

[24] F. E. Allen, Control flow analysis, in: Proceedings of a Symposium on Compiler Optimization, Association for Computing Machinery, New York, NY, USA, 1970, p. 1–19. doi:10.1145/800028.808479.
URL https://doi.org/10.1145/800028.808479

[25] Capstone engine, https://www.capstone-engine.org/, accessed: 2021-06-07.

[26] C. Linn, S. Debray, Obfuscation of executable code to improve resistance to static disassembly, in: Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS '03, Association for Computing Machinery, New York, NY, USA, 2003, p. 290–299. doi:10.1145/948109.948149.

URL https://doi.org/10.1145/948109.948149

[27] Pe format, https://docs.microsoft.com/en-us/windows/win32/debug/pe-format, accessed: 2021-06-22.

[28] Python pefile, https://github.com/erocarrera/pefile, accessed: 2021-06-22.

[29] Gnu core utils, https://www.gnu.org/software/coreutils/, accessed: 2021-06-23.

[30] Gnu core utils official github page, https://github.com/coreutils/coreutils, accessed: 2021-06-23.

[31] Mingw, http://mingw-w64.org/doku.php, accessed: 2021-06-23.

[32] Cygwin, https://www.cygwin.com/, accessed: 2021-06-23.

[33] Virusshare, https://virusshare.com/, accessed: 2021-06-30.

[34] D. C. D'Elia, E. Coppa, F. Palmaro, L. Cavallaro, On the dissection of evasive malware, IEEE Transactions on Information Forensics and Security 15 (2020) 2750–2765. `doi:10.1109/TIFS.2020.2976559`.

[35] J. Singh, J. Singh, Challenges of malware analysis : Obfuscation techniques, 2018.