# Unveiling and mitigating common pitfalls in malware analysis

Dario Nisi

**THESE DE DOCTORAT DE**
**SORBONNE UNIVERSITE**
préparée à EURECOM

École doctorale EDITE de Paris n° ED130
Spécialité: «Informatique, Télécommunications et Électronique»

Sujet de la thèse:

# Unveiling and Mitigating Common Pitfalls in Malware Analysis

Thèse présentée et soutenue à Biot, le 03/12/2021, par

DARIO NISI

| | | |
|---|---|---|
| Rapporteurs | Prof. Lorenzo Cavallaro | University College of London |
| | Prof. Christian Rossow | CISPA Helmholtz Center for Information Security |
| Examinateurs | Prof. Martina Lindorfer | TU Wien |
| | Prof. Davide Balzarotti | EURECOM |
| Directeur de thèse | Prof. Marc Dacier | EURECOM |
| Co-Encadrant | Dr. Yanick Fratantonio | Cisco Systems Inc. |

To those who did not make it through the difficult journey we shared.
You keep walking by my side, wherever I go.

# Preface

The Cambridge dictionary defines a *Ph.D.* as "the highest college or university degree." It is arguably not inspiring, but it is quite quixotic to expect any more emotions from a dictionary definition. Allow me to add a bit of a personal touch to this otherwise cold statement. As cliché as it may sound, to me, a Ph.D. is actually the end destination of a journey. At times it feels seemingly endless, tortuous, and ruthless, while at other times is exciting, empowering, and lively. During my personal Ph.D. experience, the latter drastically outweighed the former. The reason is easily explained: I have had the luck to be surrounded, supported, cheered up, fed, mentored, and inspired by some of the best people I have ever met.

First and foremost, my deepest gratitude goes to Yanick, my advisor. Thank you for always being there when I needed it with your ever-positive attitude, for transmitting me the n00b mindset (one of the most valuable life lessons I ever got), for teaching me that sometimes « it is what it is » and it is better to « wrap it up move on with our lives, » and for educating me on the importance of the Oxford comma. Oh! I have almost forgotten it; Thanks for showing me that acknowledgments in academic writings can be fun.

Special thanks go to Marc, my thesis director. While we did not collaborate much during my Ph.D. studies, I always felt you were available and ready to help when needed.

It goes without saying, I am very grateful to the defense committee, Profs. Davide Balzarotti, Lorenzo Cavallaro, Martina Lindorfer, and Christian Rossow. That of the Ph.D. jury member is often a thankless job but a fundamental one nonetheless. Thank you for taking the time to assess my research and for fueling such a rich discussion during the defense with your relevant and thought-provoking questions.

Throughout the years, I met many great professors, researchers, and students at EURECOM and, in particular, in the S3 group. I would like to thank each and every one of them, but the list would be too long. Just know

that it has been an immense honor and pleasure to work and study alongside you.

Some of them, however, I really cannot avoid mentioning, as I consider them to be family. Graziano, Luigi, Marius, Fabio, Emanuele, Savino, Frat'm Pox, Alessandro, Simone, Andrea, and Fioraldi[1]. Thanks for the fantastic time we shared.

Besides this *"foster"* family, I have to thank my real one. I would not be the person I am, have not my parents educated me the way they did and loved me the way they do. Mum, Dad, thank you for the unconditional support and for pushing me to strive for excellence. Thanks also to my brother, Giulio, my grandmas, aunts, uncles, cousins, and the rest of my extended family, as well as my all-time friends Giancarlo, Giulia, Grazia Luna, Timoteo, Greta, Barbara, Alessio, Diego, and Antonio for always being there for me.

A particular acknowledgment must also go to a few (at this stage, famous) individuals (or, should I say, "creatures"). I am, of course, referring to Betty Sebright and her team, Pino Proform, and Slasti Mormanti. Thanks for being such an endless source of inspiration. I am sure we will meet one day.

Last but by no means least important, I wholeheartedly want to thank Marlène. The sense of peace and joy you brought into my life is unparalleled. Thanks for the late-night discussions about the origin of the world, the long and accurate proofreadings, your curious questions about my research that – I admit – oftentimes cornered me, and in general for all the small and big acts of care and love with which you spoil me on a daily basis. Without your support, this thesis could have never seen the light of the day.

[1] Chronologically ordered by meeting time.

# Abstract

As the importance of computer systems in modern-day societies grows, so does the damage that malicious software causes. This led the security industry to engage in an arms race against malware authors to create better systems to detect malware and prevent it from spreading. On their side, to cope with the advances in the field of malware analysis, malware authors sharpened their tools with the objective of thwarting the analysis and defeating countermeasures. In this arms race, in fact, all wrong assumptions (no matter how subtle) may allow malware to circumvent detection systems, effectively running unopposed for a long period of time.

This thesis focuses on two aspects of modern malware analysis techniques that are often overlooked, namely the use of API-level information for encoding malicious behavior and the reimplementation of parsing routines for executable file formats in security-oriented tools. This thesis shows that taking advantage of these practices is possible on a large and automated scale. By reviewing recent evidence brought to light by security researchers and hunting malware in the wild, we also demonstrate that malware authors show increasing interest in exploiting these practices. Lastly, we study the feasibility of fixing these problems at their roots, measuring the difficulties that anti-malware architects may encounter and providing strategies to solve them.

# Résumé

L'importance des systèmes informatiques dans les sociétés modernes ne cesse de croître, tout comme les dommages causés par les logiciels malveillants. Cela a conduit le secteur de la sécurité à s'engager dans une course aux armements contre les auteurs de logiciels malveillants afin de créer de meilleurs systèmes pour détecter ces derniers et empêcher leur propagation. De leur côté, pour faire face aux avancées dans le domaine de l'analyse des logiciels malveillants, les auteurs de ces derniers ont affiné leurs outils dans le but de déjouer l'analyse et de mettre en échec les contre-mesures. Dans cette course aux armements, en effet, toutes les hypothèses erronées (aussi subtiles soient-elles) peuvent permettre aux logiciels malveillants de contourner les systèmes de détection, fonctionnant sans réelle opposition pendant une longue période.

Cette thèse se concentre sur deux aspects des techniques modernes d'analyse des logiciels malveillants qui sont souvent négligés, à savoir l'utilisation d'informations au niveau des API pour coder les comportements malveillants et la ré implémentation des routines d'analyse des formats de fichiers exécutables dans les outils orientés sécurité. Cette thèse montre qu'il est possible de tirer parti de ces pratiques à grande échelle et de manière automatisée. En examinant les preuves récentes mises en lumière par les chercheurs en sécurité et en traquant les logiciels dans leur environnent naturel, c'est-à-dire en observant empiriquement leur modus operandi au cours d'attaques réelles, nous démontrons également que les auteurs de logiciels malveillants manifestent un intérêt croissant pour l'exploitation de ces procédés. Enfin, nous étudions la possibilité de corriger ces problèmes à la racine, en mesurant les difficultés que les architectes de logiciels malveillants peuvent rencontrer et en proposant des stratégies pour les résoudre.

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

Over the last decades, the importance of computer systems in modern-day societies rose to the point of pervading every aspect of our daily life. Nowadays, computer systems provide the backbone of critical infrastructures, guard our personal information, and allow for quick transmission of essential data, inevitably affecting our private, public, and political life.

While bringing unquestionable benefits, digital technologies also opened the door to new threats that often caught users and vendors off guard. Driven by the potentially enormous gains that exploiting such a vital apparatus may deliver, ill-intentioned actors started targeting computer systems or using them to perpetrate frauds.

One common strategy employed by threat actors is delivering a piece of software that performs detrimental actions on the target system, often disguising it as harmless or even desired.

Previously relegated to the technical sphere, terms like "spyware," "ransomware," "trojan horse," and others swiftly became part of the public discourse once threat actors started employing the internet for their malicious intents. In fact, the global network provided an unprecedented vector for attack for this type of software, to the point that several authors compared the malware spread to that of an epidemic.

The parallels between the malware phenomenon and the field of epidemiology do not end with the extent of its proliferation. Similar to bacteria and viruses, also malware tends to adapt to its environment, eventually bypassing the barriers that block the infection.

However, while its organic counterparts evolve "by chance," following the rules of natural selection, malware's evolution is the byproduct of an intelligent design: Threat actors purposely equip their creations with sharper weapons and clever disguises.

Make no mistake: What ultimately makes countering malware so complex in the real world is, in fact, not only the magnitude of the malware phenomenon. The adversarial nature of this field plays an equally important role in making anti-malware research and practice so challenging and unique.

In fact, despite the security industry's attempts to promptly detect, document, and counter emerging threats, malware authors continuously improve their techniques, aiming at hiding malicious behaviors from the malware analysts' inquisitive eyes. These techniques are commonly known as evasion techniques because of their manifested objective of "evading" the analysis.

While each evasion technique is different, in one way or another, all of them exploit design flaws and limitations of anti-malware tools.

In particular, the root cause from which every type of evasion mechanism originates is that anti-malware tools must model either the system on which the malware runs or the capabilities of the malware itself. As a result, anti-malware tools reason only about an approximation of the malware and its execution. The discrepancies between these models and the reality of malware's behavior pave the way to bypassing defenses and analysis.

In all fairness, the security industry gained valuable experience during the decades-long arms race against malware. When enough new malware strains that can elude analysis arise, the industry reacts by questioning previous practices and adopting new ones to respond to the new threats.

However, this tendency for reactive approaches leads to keeping in place other practices that are known to lead to loose approximations that malware authors can leverage, at least in a theoretical or non-scalable way.

This thesis challenges design choices in anti-malware tools and techniques, adopting and promoting a proactive approach to malware analysis. In particular, we focus on two practices that are still widespread, although a fair share of practitioners considers them questionable. Contrary to the prevailing belief, we show that taking advantage of such design choices is not just a theoretical exercise but a concrete threat that the anti-malware industry is not ready to face.

While experts acknowledge that malware may exploit these practices, they argue that this is a "marginal" problem, the main argument being that these techniques are either not robust enough or require substantial manual effort: This thesis questions this argument by showing that taking advantage of these practices is possible on a large and automated scale. By reviewing recent evidence brought to light by security researchers and

hunting malware in the wild, we also demonstrate that malware authors show increasing interest in exploiting these practices. As part of this thesis, we also study the practical repercussion of these problems and the feasibility of fixing them at their roots, measuring the difficulties that anti-malware architects may encounter and providing strategies to solve them.

In more specific terms, this dissertation focuses on two aspects of modern anti-malware tools and research, ubiquitous to both malware analysis pipelines and endpoint protection software: the reimplementation of the program loader of the operating system, and the use of API-level information for capturing malware behavior.

## 1.1 Contributions

The scientific value of the research work presented in this thesis unfolds alongside three main thrusts relevant for malware analysis.

### Systematic approach to find parsing discrepancies in software handling executable file formats

A known problem in the security industry is that programs that deal with executable file formats, such as OS loaders, reverse-engineering tools, and antivirus software often have slight discrepancies in the way they interpret an input file. Attackers can abuse these differences to evade detection or complicate reverse engineering and are often found by researchers through a manual, trial-and-error process.

In this thesis, we present the first systematic analysis and exploration of PE parsers. To this end, we created a custom domain-specific language to easily capture the details on how different software parses, checks, and validates whether a file is compliant with a set of specifications. By design, models written in our language can be translated into SMT problems whose solutions are PE headers that the analyzed parser accepts as valid.

By leveraging the mathematical properties of SMT problems, we developed a framework that automatically carries out various tasks that would be hard to perform manually. For example, our framework can automatically generate samples (i.e., PE files) that satisfy a specific model. More interestingly, our framework can also produce differential test cases, PE headers that one implementation considers valid and that a second implementation would mark as invalid. As we will discuss, the existence of these discrepancies has profound repercussion for malware analysis systems.

We then used our custom language to create models for the loaders of three versions of Windows (XP, 7, and 10) and the popular tools radare2, ClamAV, and Yara. Modeling the loaders of Windows was a particularly challenging task since it required extensive reverse engineering of different operating system components, spanning from the kernel to the dynamic libraries.

By means of our framework, we compared these models and we explored the discrepancies among these loader implementations. For instance, for any combination of two versions of Windows, we were able to generate PE executables that run without problems under the first, but that the second discarded as malformed. Similarly, we generated valid samples according to Windows that the tools either marked as invalid or for which they provided an inaccurate memory mapping.

The results of our analysis have consequences on several aspects of system security. We show that popular analysis tools can be bypassed, that the information extracted by these analysis tools can be easily manipulated, and that it is trivial for malware authors to fingerprint and "target" only specific versions of an operating system in ways that are not obvious to someone analyzing the executable.

The discrepancies we found in this work represent a potent weapon in the hands of malicious actors who have strong incentives in discovering and exploiting them. To assess whether malware authors are leveraging these discrepancies in the wild, we run a malware hunting campaign on the popular malware analysis service VirusTotal. We estimate to have scanned around five million samples during this campaign, parsing their PE headers and looking for those conditions that may trigger any of the discrepancies we found. The campaign found several samples for each discrepancy, suggesting that malware authors put much effort into fine-tuning their PE executables for evading analysis and defenses.

More importantly, this work shows that the fragmentation of PE loaders' implementation poses a real problem. As we emphasize throughout the thesis, there is no one correct way to parse PE files, and even different versions of the Windows operating system treat this format slightly differently. Therefore it is not sufficient for security tools to fix the many inconsistencies we found in our experiments, but rather, they should tackle the problem at its roots.

We argue, in fact, that security tools should model accurately how different versions of the operating system parse the PE file format and prompt the user to choose which model to employ for their analysis. Our work provides comprehensive guidelines for the modeling phase, as well

as the tools to ease this process.

We explore this research thrust in a paper titled "Lost in the loader: The many faces of the windows PE file format," published at RAID 2021.

### Novel and universal API-tracing bypass technique

A recent trend that analysts reported in malware consists in implementing malicious functionalities using low-level programming primitives. While increasing the complexity of the development process, this strategy allowed threat actors to create malware that has higher chances to evade analysis and that is significantly more challenging to detect on the endpoint.

In fact, malware analysis and endpoint protection tools often rely on information encoded in terms of high-level functionalities provided by the operating system (APIs). Malware can effectively disguise its malicious behavior by relying on lower-level primitives that analysis tools do not monitor.

While effective and severe, for the time being, malware has only employed this strategy for simple tasks. This is not surprising: developing complex functionalities using low-level programming interfaces is significantly more complex, thus incurring higher development time and costs. Moreover, even when implemented successfully, relying solely on low-level programming interfaces often results in less portable programs.

In this thesis, we present an approach that remarkably eases the process of creating elaborated executables able to bypass API-based malware analysis. This research generalizes over the recent trends that analysts reported in new malware strains that directly invoke the lower-level services of the OS kernel (namely, the syscalls) in an effort to reduce their API footprint.

Our system compiles programs written using the high-level APIs of the operating system and furnishes the resulting executables with the code implementing the APIs it needs, derived by the original Windows libraries. At compilation time, our system resolves the program's dependencies and finds the DLLs that it requires. The DLLs are then modified to allow the program to use their functionalities without the need to be loaded by the operating system and joined together to create a custom runtime environment.

At execution time, the programs use the functionalities provided by the embedded runtime instead of the system libraries. By doing this, the program is completely transparent from the point of view of any API-

monitoring solution. These tools rely on the assumption that for a program to use an API, its execution flow has to reach the code provided by the system libraries. However, programs compiled with our approach never need to step into system-provided (thus, potentially monitored) modules, and can adopt arbitrarily complex obfuscation techniques to hide the nature of the copy of the DLLs that they include.

What makes our technique potentially game-changing is the minimal effort that malware authors have to put in place to adopt it. In fact, our framework can be plugged into a typical development workflow without rewriting or modifying the malicious logic.

We explore this research thrust in the first part of a paper titled "Generic and Practical API Tracing Bypass for Windows Malware" (currently under submission).

## Characterization of the semantic gap between low-level and high-level programming interfaces

A common and generic way to encode the behavior of a program is to track which APIs (Application Programming Interface) it employs. Modern operating systems provide these highly specialized programming interfaces that ease the development of complex functionalities used by benign and malicious software alike.

API-based encoding of malicious behavior has proven effective for malware analysis due to the high level of information and the rich semantics that each API carries.

However, while many previous research works, malware analysis tools, and endpoint protection solutions rely on high-level information to encode malicious behaviors, collecting such information in a reliable and stealthy way is unfortunately impossible. Moreover, techniques such as the one we discussed previously can create malware that does not rely at all on the API layer for performing complex tasks.

The common analysis technique to address this concern is to move past an API-centric conception of behavior analysis, and focus at "the syscall layer" as the best lookout post for reliably monitoring programs' runtime behavior, since it is more difficult, if not outright impossible (depending on the threat model) to bypass.

However, employing syscalls primitives as the basic blocks for encoding a program behavior does not come without a price. Indeed, syscalls carry significantly lower semantics than APIs, which makes them more challenging for human analysts and even automated tools.

Moreover, some aspects of program execution may not be accessible at all when choosing to encode its behavior in terms of syscalls. Some functionalities that programs often use do not need the intervention of the operating system to be completed. Notable security-sensitive examples are encryption and decryption routines which only require mathematical operations and, as such, can be carried out in userspace.

In this thesis, we set out to explore "the semantic gap" between API- and syscall-level information in two modern operating systems, Windows and Android: how "far" is the information provided by syscall traces with respect to their API-level counterparts? If syscall traces appear to not be as legible as API traces, what is the scale of the problem? semantics-wise, how big is the gap between these syscall and API traces? An how challenging is it to develop an approach to automatically "bridge this gap"?

For what concerns the Windows operating system, we report the results of a large-scale dynamic analysis campaign, which characterizes the complexity of the semantic gap between the WinAPI and the NativeAPI layer (roughly equivalent to the syscalls).

For each of the over 23 thousand programs in our dataset, we collected both the API- and syscall-traces employing a custom dynamic analysis tool based on dynamic binary instrumentation. We then analyzed the collected data to understand which aspects of a program's behavior can be reconstructed from a syscall trace and the feasibility of recovering the API-level information. Unfortunately, our measurements on the collected data show that many factors make the syscalls-to-APIs mapping problem ambiguous, turning it into a practical challenge.

In a similar fashion, we measured the feasibility of reconstructing API-level information from syscall trace, this time targeting the Android operating system.

Similarly to what happens in the Windows ecosystem, also in Android, the vast majority of existing frameworks perform API-level tracing (i.e., they aim at obtaining the trace of the APIs invoked by a given app), and use this information to determine whether the app under analysis contains unwanted or malicious functionalities.

Driven by previous work that showed that, in Android, API-level tracing and instrumentation mechanisms could be easily evaded, regardless of their specific implementation details, in this chapter, we tackle the mapping problem between the Java API-layer of Android and the Linux syscall layer onto which it builds.

The first part of our approach consists in collecting runtime information from a dataset of Android apps that allows inferring the caller-callee

relationships between APIs and syscalls.

To this end, we developed a novel dynamic analysis system capable of tracing both Java APIs and syscalls that an Android app invokes during its execution.

To build the analysis system, we relied on automatic source code analysis/editing techniques to provide the Android operating system with a custom logging facility. In particular, we instrumented each Java method's entry and exit points in the Android source code to emit a custom message that the logging interface then captures.

To ease the process of analyzing the massive amount of data collected during the dynamic analysis phase, we created an easy-to-query data structure that stores information about each recorded API, the knowledge base.

To condense the information stored in the knowledge base, we created models for each API we recorded. Intuitively, these models function as regex-like objects that match all the possible syscall traces that the invocation of an API can produce. As such, these models can be used to find potential evidence of API invocation from a syscall trace. However, our attempts to reconstruct the API information from the syscall traces we recorded during the dynamic analysis were unsuccessful due to the inherent similarities among the models of different APIs.

In an attempt to characterize the root causes, first by manually investigating the knowledge base and then by employing an automated approach, we discovered several syscall patterns that appear during the invocation of many different APIs in a seemingly non-deterministic manner. After a more in-depth investigation, we concluded that these patterns originate from synchronization and memory management primitives that many APIs use. For this reason, we consider these syscall patterns as noise since they do not convey any valuable information about the API that invoked them.

Noisy patterns significantly contribute to the ambiguity of the semantic gap between the API and the syscall layers. However, even removing these noise sources, solving the mapping problem seems to remain out of reach. Even under strong, favorable assumptions, the syscalls sequences that different APIs produce appear too similar to be distinguished reliably.

For what concerns Windows, we explore this research thrust in the second part of the paper titled "Generic and Practical API Tracing Bypass for Windows Malware" (currently under submission); for what concerns Android, we explore it in a paper titled "Exploring Syscall-Based Semantics Reconstruction of Android Applications,", published at RAID 2019.

## 1.2 Thesis Outline

The thesis is organized as follows. Chapter 2 provides the basic knowledge required to understand the contribution in this thesis.

Chapter 3 is based on the paper "Lost in the loader: The many faces of the windows PE file format" published at the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2021), and systematizes our analysis of the PE loaders in the Windows ecosystem.

Chapter 4 is based on a paper titled "Generic and Practical API Tracing Bypass for Windows Malware" (currently under submission) and presents our generic bypass of API tracing as well as a characterization of the semantic gap between APIs and syscalls, focusing on the Windows operating system.

Chapter 5 presents our attempt at reconstructing the API semantic from syscall traces recorded from Android applications and is based on the paper "Exploring Syscall-Based Semantics Reconstruction of Android Applications," published at the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019).

Finally, Chapter 6 concludes the summary and provides possible future research directions.

# Chapter 2

# Background

Given the magnitude of the malware problem, it is not surprising that researchers and practitioners have developed many tools and approaches to counter it.

Since the outset of the malware phenomenon, the security industry has implemented a two-stage strategy to mitigate its impact and protect the end-users. In its early days, malware analysis relied heavily on manual effort. Analysts would inspect potentially malicious samples, gather insights about their behavior when executed, and report its distinguishing features if harmful. These features are then translated into actionable procedures — often called "signatures" — that scan files and programs on the user machine, removing those that match the malware footprint.

While this two-stage model remained in place until now, the technologies employed to analyze new malware and detect it on user equipment evolved.

On the analysis side, the most significant breakthrough happened with the introduction of automated pipelines that generate reports about several aspects of program execution, such as network and filesystem operations and modifications to the system settings. Automated systems allowed malware analysis to scale up and keep pace with the ever-increasing number of samples discovered every day. In fact, malware analysts could now focus only on those samples for which the automated tool reported something worth investigating, reducing the overhead of auditing completely benign ones.

The endpoint protection stage also witnessed new developments over the years. Malware signatures recognition tools started as simple byte-patterns scanners but eventually adopted more powerful technologies, such as domain-specific languages, to perform more complex checks.

Eventually, anti-virus companies introduced dynamic signatures (that reason about the malware at runtime) and telemetry services to collect statistics about the user machine from which carving evidence of concealed attacks.

This chapter will dive into the technical details and the common implementation choices of malware analysis and detection techniques. As we will see, to counter malware effectively, one needs extensive knowledge of the underlying operating system principles; After all, malware usually comes in the form of "regular" executables meant to run under a specific operating system. For this reason, we also provide background information about those aspects of modern operating systems design that are particularly relevant for themes tackled in this thesis. In particular, we detail the typical characteristics of the programming models that modern operating systems adopt, as well as the basics of executable file formats.

## 2.1 Executable File Formats

One of the main tasks of an operating system is to set up the environment for user programs to run correctly. This preparatory stage is often referred to as "loading" and entails creating a new process, arranging its address space the way the program expects it, mapping in memory the modules on which the program relies, and eventually let the execution begins, starting from the program's entry point.

Naturally, the loading process is not the exact same for each program. Indeed, the operating system needs to fine-tune the different stages to meet the detailed requirements of each program. *Executable file formats* provide the instruction to encode such requirements into data structures stored within the program's file.

In most cases, these file formats are designed to accommodate the various features of the operating systems that support them. This is the reason for which virtually each major operating system provides support for only a handful of executable file formats. Just to name a few examples, nowadays, the Windows operating system only supports the *Portable Executable* file format, Linux supports the ELF format, while the format of choice for modern macOS systems is Mach-O.

On top of playing a significant role in operating system design, executable file formats represent a major ally for malware analysis. The metadata stored in a program's headers, in fact, proved very valuable for malware analysis. For example, by parsing a program's headers, anti-malware tools can recover its code which can then be analyzed using static bi-

nary analysis techniques. Moreover, in operating systems that support dynamic linking, the executable file formats encode which external module and specific functionalities the program relies on. In the context of malware analysis this information hints at the program's intent: Certain modules and functionalities are more commonly employed by malware than regular non-malicious software.

### The PE File Format

A significant portion of this thesis focuses on analyzing the ecosystem around *Portable Executable* (PE), and it is worth providing more in-depth information about this executable file format.

The PE format is the standard executable file format supported by the Windows operating system family [pe ]. Adopted by Microsoft since the release of Windows 3.1, this format underwent a series of revisions throughout the years that added support for new features. However, its core design remained unchanged and consists of a number of structured data types, commonly called "headers."

Figure 2.1 shows the structure of a *PE* executable. The first header at the beginning of the file is the *MZ Header*, originally used in the MS-DOS operating system and still in use today for backward compatibility. For Windows-specific executables, the *MZ Header* is only used for determining the offset at which the *COFF Header* starts. This second structure contains essential information about the executable, including the architecture on which it is meant to run, whether it is a dynamic library or a standalone executable, and whether its image supports a randomized base address.

The *Optional Header*, which starts right after the *COFF* header, provides more detailed information, including the preferred *ImageBase* (i.e., the virtual address of the first byte of the image in case no relocation is applied), the *SizeOfImage*, and the amount of memory to reserve for the stack and heap. The peculiarity of this header is that its size is not fixed but rather determined by the *SizeOfOptionaHeader* in the *COFF Header*. This design choice makes the *OptionalHeader* easy to extend in future revisions of the format specifications. Other fields of the *Optional Header* worth mentioning are the *Subsystem*, *MajorSubsystemVersion*, and *MinorSubsystemVersion*. The former indicates the Windows Subsystem required to run the executable (e.g., a program using the graphic user interface requires the Windows GUI Subsystem). The other two specify the minimum version of the Subsystem required. Similarly, the *MajorOper-*

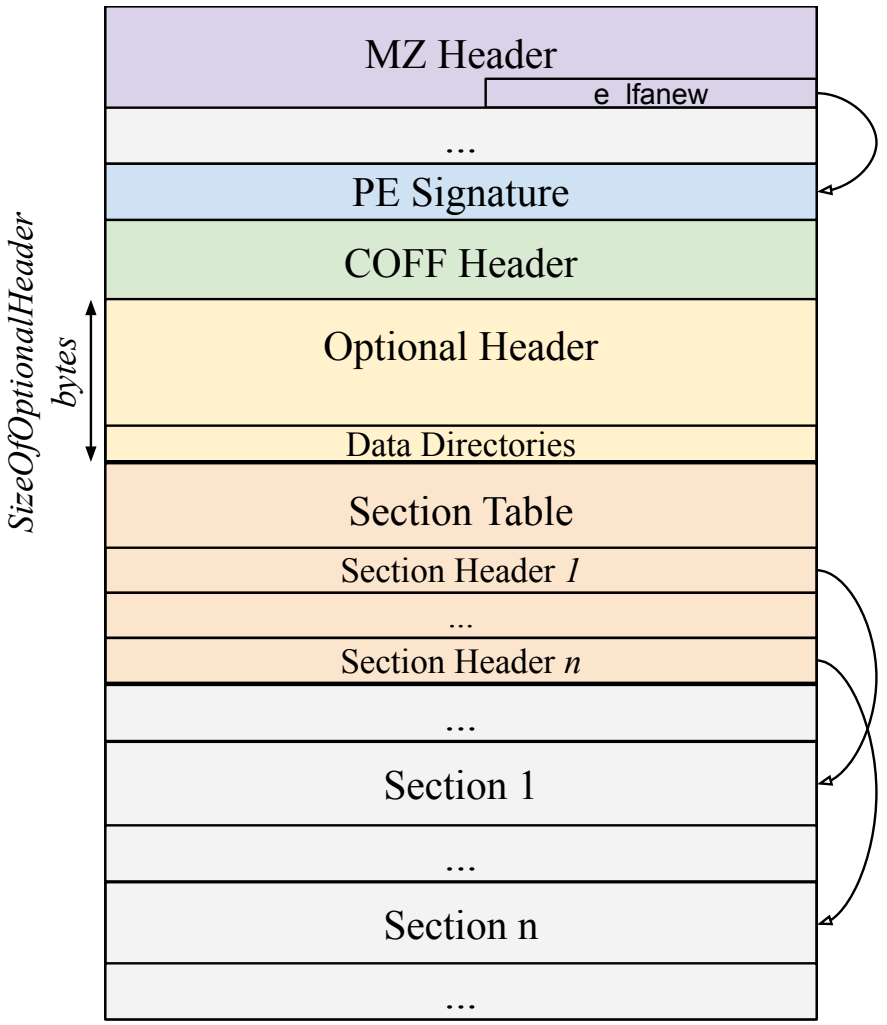


Figure 2.1: Structure of a PE Executable

*atingSystemVersion* and *MinorOperatingSystemVersion* fields specify the minimum version of the operating system required to run the program.

The last portion of the *Optional Header* contains the *DataDirectory* table. A *DataDirectory* contains the relative virtual address (i.e., the offset from the base address of the memory image; from now on, *RVA*) and the size of an additional data structure used for multiple purposes. Examples of *DataDirectories* are the *Import Table*, which declares the dynamic libraries and the functions that need to be loaded alongside the executable; the *Export Table*, containing the functions that the executable makes available for other programs to use; the *Relocation Table*, which provides a set of instructions on how to patch the executable in memory when it is not loaded at its preferred base address; and the *Certificate Table* that contains the digital signatures of the developer of the executable. The number of *DataDirectories* is not fixed and it is determined by a field in the *Optional Header*.

The *Section Table* starts at the end of the *Optional Header*. Each entry in this table defines a section, i.e., a contiguous memory region of the image, either uninitialized or populated with portions of the executable. Each section has its name and characteristics, such as the memory access permission level or whether the operating system can swap out its pages in case of a memory shortage. Sections logically organize the executable into homogeneous portions. For example, by convention, the *.text* section contains the program's code, while the *.bss* contains the uninitialized data.

## 2.2 Programming models in Modern Operating Systems

Modern operating system designs define a strict and clear boundary that separates the kernel-space from the user-space. While the latter hosts most of the applications running on the system, the kernel-space is reserved to the critical portion of the operating system (not incidentally called the *kernel*).

While this separation prevents single applications from harming the entire system, either by malice or by accident, it also drastically reduces the capabilities granted to each application running in user-space.

For this reason, modern general-purpose operating systems provide one or more *Application Programming Interfaces* (APIs), sets of built-in functionalities provided by the operating system that would otherwise be very tedious, error-prone, or even downright impossible to implement in

the context of a user-space program.

In particular, the Windows operating system provides two distinct sets of *APIs*: [1] *WinAPIs* and *NativeAPIs*. The first represents the higher-level layer, providing complex functionalities while hiding implementation details to the user program. Examples of *WinAPI* are those functionalities related to networking (including network protocol implementations for standard protocols and cryptography), the graphic user interface, and system services management.

*NativeAPI*, instead, is the lightweight and lower-level layer upon which the *WinAPI* relies. With few exceptions, most notably the C runtime library and the user-space loader, the actual implementation of the majority of the *NativeAPIs* resides in the kernel, which exposes them as system routines that the user-space program can access using special CPU instructions (e.g., `int`, `sysenter`, and `syscall` in the Intel x86/x86-64 ISA). In the remainder of the thesis, we will indicate this subset of *NativeAPIs* with the generic term *syscalls*. Lower-level modules often depend solely on the *NativeAPI* layer. Such is the case of device drivers (that need to interact with the kernel directly) and those OS components that participate in the early initialization phases of the system startup, i.e., a moment in which the *WinAPIs* are not available yet.

The degree of their complexity, however, is not the only difference between *NativeAPI* and *WinAPI*. Indeed, while the latter is guaranteed to be stable among different major releases of the Windows operating system, the latter may change drastically. Introductions and removals of *NativeAPIs* happened various times throughout the years, as documented by [j00], which also showed that even the application binary interface for the same syscall (in particular, the so-called "syscall number") changed among different releases of the same major version of the operating system. Moreover, while *WinAPIs* are thoroughly documented, some *NativeAPIs* are not, as they are intended for internal use only.

Although conceptually, it would be possible to write any program using either the *WinAPI* layer or the *NativeAPI* one (after all, the former builds on top of the latter), the choice is straightforward from a programmer's perspective: By providing better portability with less implementation complexity, the *WinAPI* layer is inevitably the most cost-effective in the vast majority of circumstances.

For many aspects, the programming model of the Android operating system is similar to its Windows counterpart. The so-called *Android API* is

[1] In this thesis, we consider the *API* acronym as countable, indicating any of the functions that provided by the operating system. The same applies to *WinAPI* and *NativeAPI*.

the most abstract of the three layers and can be accessed through the Java and Kotlin programming languages. Android APIs provide a wide range of functionalities, from networking to cryptographic primitives and user interface management.

The Android operating system also provides a lower-level programming interface, allowing developers to write portions of their applications using the C++ language and compiling them into *native code*. The support for native code was originally introduced to free performance-critical components of applications from the burden of using a managed language, at the expense of portability.

From native code, an application can virtually access any high-level Android API. In fact, the Android framework itself leverages native code to implement any functionality that requires the intervention of the kernel, including the most security-relevant APIs. For example, the APIs that handle the user's personal data, interact with the broadband and access the Internet are all implemented in native code.

Similar to the case of Windows, in Android, the least abstract programming layer is the *syscall layer*, on top of which the *Android APIs* and the *native APIs* are built. Unlike Windows, however, the syscalls in Android are relatively stable throughout the different operating system versions, a property inherited from the Linux kernel.

### Encoding Malware Behavior

The programming model of an operating system and its APIs play a fundamental role in program analysis in general and in malware analysis in particular. In fact, the sequence of APIs invoked by a program (together with their arguments) is often used for encoding its runtime behavior.

As detailed previously, modern operating systems usually provide a layered programming interface, allowing programs to access either high-level semantic-rich APIs, as well as low-level ones.

Choosing which layer of the programming interface to use for encoding a program's behavior inevitably affects the expressiveness and the semantics richness of the encoded information. For example, describing the execution of a Windows program in terms of *WinAPIs* will result in more intelligible information that professionals with minimum experience in Windows programming can understand. The same cannot be said if *NativeAPIs* were to be used to encode the same execution because the semantic that this layer carries is significantly lower. Moreover, the lack of documentation for *NativeAPIs* and *syscalls* makes interpreting this type of

information even more complicated.

As we will explain in more detail in Chapters 4 and 5, however, obtaining high-level information by directly observing a program's execution is unsafe and can ultimately lead to evasion strategies.

Moreover, previous works have often completely neglected lower-level programming interfaces and focused only on high-level ones. For example, Afonso et al. [ABF+16] pointed out how several static analysis tools for Android applications only collect and process *Android API* information, completely disregarding *native code* components.

## 2.3 Dynamic Malware Analysis and Detection Techniques

Understanding the behavior of a program is a vital step toward determining whether it is malicious or not, which in turn is paramount for both malware analysis and endpoint protection. Dynamic analysis aims at gathering this information from running the program in a controlled environment, recording as much evidence of malicious activities as possible.

Depending on the type of controlled environment, the collected information can vary. Execution traces are one of the most common types of evidence collected during dynamic analysis, and they describe a timeline of what was executed in the context of the program under analysis.

For what concerns the Windows ecosystem, the security and reverse-engineering community has proposed several dynamic analysis techniques to trace a program's execution in terms of *WinAPIs*, which are the higher-level programming interface for the Windows operating system, as we detailed in Section 2.2. Choosing to encode a program's behavior in terms of *WinAPIs* provides remarkable advantages in terms of the intelligibility of the analysis results over *NativeAPIs*.

We can categorize the proposed techniques into three main groups based on where the instrumentation resides.

**User-space techniques** achieve tracing capabilities by modifying the process address space of the program under analysis. The simplest of such techniques is *API hooking*, which introduces instructions at the beginning of the code implementing each API, diverting the execution flow towards the logging component each time the sample under analysis invokes any API. Despite being old, *API hooking* is still widespread among EDR solutions and sandboxes. For example, the Comodo OpenEDR [Com] solution implements *API hooking* based on the *Microsoft Detour* instru-

mentation framework, while the open-source sandbox Cuckoo [OM13] ships its own hooking engine. Another prominent user-space technique is dynamic binary instrumentation (DBI). Tools such as Intel Pin [Int] and DynamoRIO [BGA03] allow an analyst to instrument a program execution by interleaving its original code and user-defined routines. Previous works [CMF+18] also used user-space techniques for import table reconstruction by hijacking the library loading process to provide the program under analysis with a custom monitored version of the requested library.

**Out-of-guest techniques** preserve the malware address space by moving the instrumentation logic outside its process' context. These techniques may rely on either ISA emulation or virtualization to create a fake and instrumented machine into which to run the program. An example of emulation-based instrumentation is PyREBox [Cisd], which traces API invocation by monitoring the *jmp* and *call* instructions the program executes. If the target address of these instructions belongs to a system-provided API, the tool adds it to its report. Virtualization-based tools like Drakvuf [LMP+14] work similarly to their emulation-based counterparts, but they leverage hardware virtualization primitives for performance and isolation reasons.

**Hardware-assisted tracing** employs the tracing facilities that the CPU provides (e.g., Intel BTS) to record each branch instruction that the processor executes. Techniques based on hardware-assisted tracing have been employed for import table reconstruction [CML+21].

Similar to its Windows counterpart, the landscape of dynamic analysis techniques for Android is quite diverse. Also in this case we can identify three main strategies to capture a program's behavior in terms of the high-level primitives that the operating system provides (namely, the *Android APIs*).

**Ahead-of-Time compilation modifications** techniques were introduced after Android switched from its legacy bytecode-oriented design (Dalvik) to the newer and more performant design called ART (Android Runtime). In the new design, Android apps are still shipped to the device in bytecode form. At installation time, however, the Android system performs a translation from bytecode to native code, adopting an "ahead-of-time" (AOT) approach. Previous works [BBS+17, SWL16] have leveraged the AOT translation process to instrument the application, introducing monitoring logic.

**Runtime instrumentations** also allow gathering information about the

APIs that an app invokes. Frida [fri] achieves runtime instrumentation by injecting a Javascript engine into the application address space. This engine can then interpret user-specified instrumentation scripts, written in Javascript, and allows, among other things, to register callbacks that are executed each time the app's execution reaches specific functions on the Android framework. Runtime instrumentation can also be achieved by modifying the Android operating system directly, like in the case of Xposed [Dev17] and Magisk [top].

**Emulation-based introspection** employs modified versions of the Android emulator to trace the application to analyze. Dynamic analysis tools such as DroidScope [YY12] and DECAF++ [DQQY19], and malware sandboxes like the Android Malware Sandbox [Are] adopt this approach.

# Chapter 3

# Finding Parsing Discrepancies for Executable File Formats

***A Systematic Exploration of the Portable Executable File Format and its Ecosystem***

## 3.1 Introduction

Over the past thirty years, malware authors have developed many techniques to bypass both static and dynamic analysis tools. The goal of the attackers is to either bypass or reduce the effectiveness of malware analysis tools while still producing samples that the target system would correctly execute.

For example, in a recent study conducted by Cozzi et al. [CGFB18], the authors discovered that malware authors often tamper with the executable file format to obtain binaries that are executed correctly on the target device but are discarded as malformed by the vast majority of the analysis tools (including disassemblers, debuggers, and common utilities to inspect the file headers). Along the same line, in 2017, Kim et al. [KKD17] discovered a set of problems in the way AV products parse and validate signed PE files. Specifically, the authors noticed that if malicious files contain the Authenticode signature copied from a benign application, they are not analyzed. Even worse, many AV engines saved time by not even scanning signed binaries at all.

While these studies point out interesting discrepancies, we believe these findings are just the tip of the iceberg of a much deeper problem:

while the inner structure of executable file formats is defined and generally well understood, the way these files are parsed and validated differs significantly among tools and operating systems and, surprisingly, also among different components of the same operating system [J. 13].

Today, the security industry employs a completely automated malware collection, analysis, and classification process to handle the massive number of new samples discovered every day. This relies on a complex toolchain that combines many components to extract static features, collect the runtime behavior from malware analysis sandboxes, and compare each sample with information extracted from similar programs. Sadly, all the existing infrastructures rely on a subtle and often overlooked assumption, i.e., *that every single component should parse, understand, and validate the sample in the same way*. Ideally, the task of parsing the executable file format should be delegated to a common standard library used by all components. In practice, instead, every program implements its own parsing and validation routines, resulting in a multitude of strategies that often differ in many relevant details. Even worse, these techniques are not the same as those adopted by the operating system to decide whether a binary can be correctly loaded in memory and executed. On the one hand, these differences may result in samples that are erroneously discarded (because they are considered malformed by some static tools) or only partially analyzed. On the other hand, as measured by Ugarte-Pedrero et al. [UPGB19a], it can result in the fact that a large number of damaged files are still assigned to dynamic analysis sandboxes — thus wasting a considerable amount of time and precious resources for security companies.

In the past, researchers have tried to look at this problem by partially documenting to what extent the structure of a PE file can be manipulated without compromising its functionality [Ale] or by collecting notes on some parts of the Windows loading process [Tod17]. However, these studies followed a simple trial-and-error approach that failed to capture the scale and complexity of the problem. In fact, previous attempts often resorted to fuzz the file header fields to test whether the resulting file could still be executed in the system. However, this approach does not account for possible inter-relationships between fields (in which different parts of the file need consistent information) and therefore provided limited results.

The goal of this chapter is to shed light on this complex problem by performing a comprehensive analysis of the factors that affect the parsing of the PE executable file format and on the fields that are read and used by different software and operating systems. An overview of our contribu-

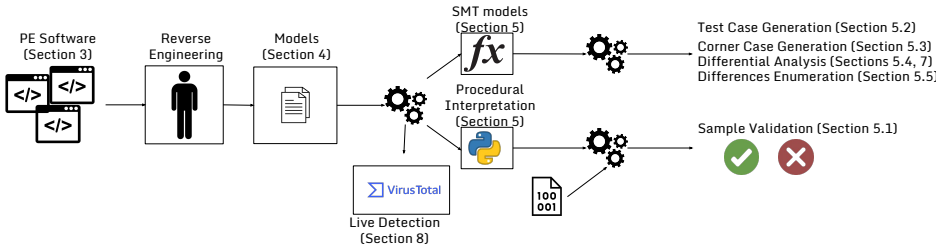


Figure 3.1: An overview of our analysis process.

tions is depicted in Figure 3.1. In particular, to perform our systematic exploration, we developed a new framework to precisely describe the steps commonly performed by OS loaders and file parsers. While this chapter focuses on the PE file format, the framework is generic enough to support the description for other formats. We started our analysis by focusing on the OS loaders used in different versions of Microsoft Windows: Windows XP, Windows 7, and Windows 10. For each version we wrote a model that lists the checks and operations that are performed to determine 1) if the file is a valid PE and thus should be "loaded" in memory and 2) how the loading process extracts and parses information from the file. We also focused our attention on different categories of security-related software that deal with PE files, such as reverse-engineering tools and antivirus programs.

An analyst can use our models to perform several different tasks, such as sample validation, sample generation, corner case generation, differential analysis, and differences enumeration. Thanks to these fully automated techniques, our framework was able to *systematically* enumerate the discrepancies that exist between the Windows loaders and popular reverse engineering tools. Our findings have significant repercussions.

First, we show how popular analysis tools can be completely bypassed and how the information they extract can be easily manipulated "at will." We also ran a VirusTotal LiveHunt session and discovered that real-world malware is currently abusing some of these (previously unknown) discrepancies in-the-wild. These findings highlight how the research community lags behind malicious actors, thus making our systematic exploration of these aspects even more timely and important.

Second, we show that the differences in which the three versions of Windows parse PE files allow attackers to create targeted executables that would run on a specific version but would be discarded as malformed by others. This could be used, for example, to evade common malware anal-

ysis sandboxes, which often run Windows 7. However, by far, the most important consequence of the differences among OS loaders is that not only the PE standard fails to describe how an executable should be parsed and validated, but also that *a de-facto reference implementation does not even exist*. Instead, our experiments show that there are as many ways to interpret a PE file as there are versions of Windows, and therefore security tools should decide which model to use on a case-by-case basis. In other words, we show that since there is not a single correct way to parse PE files, fixing security tools is significantly more complex than just "fixing bugs." Instead, we believe that the only way to tackle this problem at its root is to offer the possibility to select *which* of the several loaders a tool should mimic, in order to adapt the validation and the extracted information to the system under analysis.

In the spirit of open science, we release the entire source code and datasets produced for this work at https://github.com/eurecom-s3/loaders_modeling, and we hope this will inspire a community-driven effort to refine our models and to extend them to different file formats.

## 3.2 A Critical Look at the PE Specifications

Before diving into the specifications of the *PE Format*, it is useful to introduce some terminology that we will use in the rest of the chapter. We define the term "PE executable" (or simply "executable") to mean a file that follows the specifications of the *PE Format*, while we call "Process Image" (or simply "image") the representation in memory of the executable after it is loaded. In the remainder of this chapter, we also use the term "parser" in an informal way to refer to the general activity required to load the data contained in a PE file and map it to a set of predefined data structures. Finally, we call "validation" the process required to verify whether the information in a PE file satisfies a set of structural and logical constraints. As explained in Section 3.3, different classes of applications may parse and validate PE files in different ways and may take different actions when such constraints are not satisfied.

In addition to defining the structures of the headers (as described in Chapter 2.1), the *PE Format* specifications introduce "constraints" on their fields. With this term, we indicate a set of conditions that the fields must respect to be considered "acceptable." Trivial examples of constraints are that the first fields of the *MZ* and *COFF Headers* must contain their respective *magic numbers*: MZ and PE.

Other constraints are instead more subtle and complex. For example, each entry in the Section Table is expected to start at a multiple of *SectionAlignment* and populated with portions of the executable starting at an offset multiple of *FileAlignment* (both these are fields of the *Optional Header*). Moreover, they are expected to be sorted in ascending order by their starting virtual address, and adjacent entries in the table must be contiguous.

However, what the *PE Format* specifications fail to convey is what happens in case an executable does not meet such constraints. In other words, the specifications do not address the following questions "What happens if a section does not start at a multiple of *Section Alignment*?" or "Can a PE executable whose sections are not sorted be considered valid nonetheless?"

Other ambiguities in the specifications concern the concepts of "default values" and "architecture-specific" features. The *SectionAlignment* field itself is an example of the first category. According to the specifications, the "default value" of this field is the page size in bytes of the architecture (e.g., 4K for Intel x86). Considering that sections are also used for specifying the memory access protection level of the corresponding memory regions, it is reasonable to assume that the value of the *Section Alignment* should be at least as big as the page size. In fact, a value of *SectionAlignment* smaller than the page size of the architecture could prevent the operating system from correctly enforce memory access permissions in adjacent sections.

OS loaders could handle this corner case in different ways. A strict loader could discard any executable with a value of *Section Alignment* smaller than the page size, while a more permissive one could still load the executable but without enforcing the access permissions of the sections. The important aspect is that the PE specifications do not provide any guidance about this implementation choice and provide no insights into handling this and similar other cases.

Finally, some relocation types described in the specifications fall within the "architecture-specific" features. For instance, the possible values of the *Base Relocation Type* enumerator are "meaningful" only in certain architectures. The concept of "meaningfulness," however, is not better elaborated in the specifications leaving, again, plenty of room for different implementation strategies. Should a programmer developing software that parses the *PE Format* consider relocation types that are "non-meaningful" for the targeted architecture as a violation of the specification (and interrupt the parsing of the file) or simply ignore them?

To summarize, the *PE Format* specifications do not clearly specify all the circumstances under which a file should or should not be considered a valid PE executable, nor do they provide unequivocal guidelines on how to handle corner cases. This, alongside the large amount of software that needs to deal with PE files, leads unavoidably to discrepancies in how PE executables are handled.

## 3.3 Software Handling PE Files

There are many classes of software that need to operate on PE files, for different reasons and with different objectives. In this section, we explore some of these classes, by grouping them according to their purposes and by discussing what are the basic operations that each group must perform to carry out its tasks.

### 3.3.1 Basic Operations on PE Executables

We identify three main operations that are performed by software that deals with the *PE Format*. Note that we call these "operations" and not "phases" because, as we found out in our work, they are usually interleaved and not implemented as self-contained, sequential stages in the software workflow.

**Structural Checks.** Operations of this type ensure that the file respects the basic structure of the *PE Format*. For instance, tests to verify the magic numbers or that the offset of a data structure points within the file boundaries are examples of structural checks. Depending on the purpose of the software, these checks can be either strictly or loosely enforced. For example, some programs adopt a best effort approach and focus on gathering as much information as possible from the PE headers. For this reason, they might not abort the entire process if a structural requirement is not met, instead preferring to continue the process by focusing only on those parts of the headers that respect the PE structure. Other software may implement instead a more rigid structural validation, thus rejecting the files in which a structural check fails. Even within the same piece of software some structural checks can be enforced more strictly than others. This is because not all PE headers play the same role in all the software implementations of the format: Some may be essential, while others may be used only for optional features.

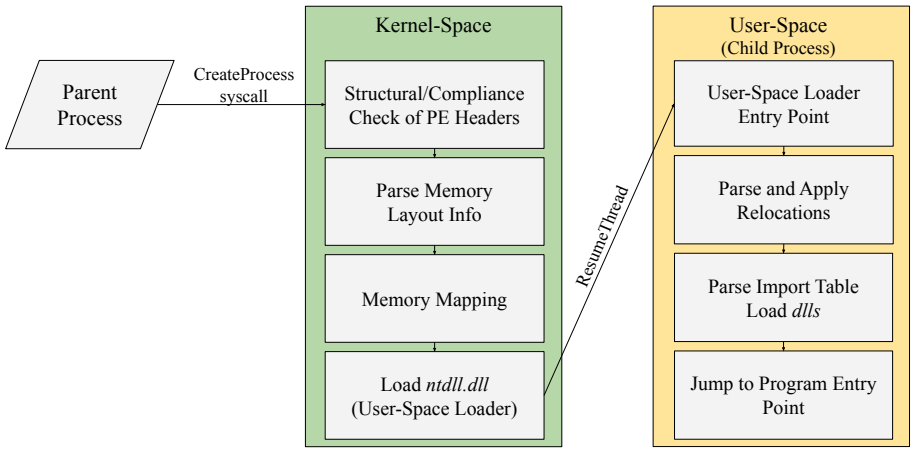


Figure 3.2: Windows Loading Process

**Compliance Checks.** This type of operation ensures that the PE executables conform to both software- and architecture-specific constraints. For instance, the *Machine* field of the *COFF Header* specifies the CPU type on which the executable is meant to run. Operating systems may use this field to determine whether a program can be executed on the current CPU architecture. Compliance Checks are usually strictly enforced, and a violation of their constraints often results in rejecting the file.

As a rule of thumb, structural checks verify if a file is well formed, while compliance checks test whether the extracted information is valid and can be used to perform the task of the software. However, the boundary between *structural* and *compliance checks* is not always clear because of the ambiguities in the specifications of the *PE Format* that fails to declare the structure of a PE executable in a formal way. Despite this difficulty, we believe the distinction among the two types of checks can ease the discussions of the various aspects of our work.

**Memory Mapping.** These operations use the gathered information to prepare the *process* address space by creating a memory representation of the PE file suitable for execution. This image includes memory areas initialized with data extracted from different portions of the executable, memory areas mapped in the process address space but left uninitialized, as well as a description of the memory access permissions for each memory areas.

### 3.3.2 PE Software Landscape

Many classes of software need to parse PE files, and for this work we focus on three main categories, which we selected because they play a particularly important role in the security domain and because we believe they exemplify the different types of operations described above.

**OS Loaders.** Their objective, as their name suggests, is to load the image of the program in memory. In the case of the Windows operating system, the "PE loader" is not a well defined, self-contained component. As Figure 3.2 shows, there are two distinct parts that contribute to the process of loading PE executables in memory. The first one is embedded in the Windows Kernel and allocates the memory for the new process, as well as the kernel structures that identify it. This stage of the loading process also loads the *ntdll.dll* library in the new process. Once this first component finishes its tasks, the execution of the new process starts from a function in *ntdll.dll*, which initiates the second part of the loading process.

Both stages enforce *structural* and *compliance checks* on the PE headers and the structural checks are usually enforced strictly on all the headers. Once the loader has gathered the information it needs, it proceeds to allocate and populate the memory needed by the program. In other words, operating system loaders need to perform *all* the three basic operations described above. This, alongside the fact that they are implemented in two different components, makes this class of software the most complex and challenging to analyze.

**Reverse-Engineering Tools.** Programs in this class are used for software binary analysis. For this purpose, they need to gather information from the PE headers to recreate a memory representation similar to what an operating system loader may produce. This means that they are mostly interested in *memory mapping* and often perform *structural checks* with a best-effort approach. For example, they might rely on the debug information stored in the *Debug Header* if it is available, but fallback on other heuristics if such header is not present or is malformed. On the other hand, to be able to analyze as many executables as possible, this class of software performs very few *compliance checks*.

**Antiviruses.** Software in this class performs both *structural* and *compliance checks* on the information gathered from the PE headers of the executable. These constraints are meant to ensure that the executable provides the data needed for their format-specific signatures. However, they may perform very little *memory mapping* operations, usually only to en-

able signatures to convert *RVAs* to offsets into the original executable file (e.g., the *rva_to_offset* function in *yara* [yarb]).

## 3.4 Constraints Modeling

As we discussed at the beginning of the chapter, researchers have already documented several inconsistencies[A. 13, BB13] in the way different programs parse PE files over the past years. These previous attempts have focused on black-box approaches that try to construct anomalous files and observe whether a given OS (or analysis tool) would process them correctly. Altough automated black-box approaches, e.g., fuzzing, can find interesting differences, they either could only do that in simple cases or without the ability to list all possible differences exhaustively. Moreover, as it is the case for software testing, whenever such approaches are unable to find differences, it is not possible to conclude that two applications process PE files in an equivalent way.

For this work, we are interested in finding the same flavor of behavioral differences, but we take a completely different approach: we opted for translating the parsing procedure of the various applications into formal models, which we can then query to explore the space of "acceptable inputs" *systematically*. Our observation is that both structural and compliance checks, as well as memory mapping operations, can be ultimately deconstructed into a set of formulas and constraints over the fields of the PE structures.

Our main goal is thus to develop models to describe those formulas and constraints in a way that can be used in an automated fashion to 1) compare different models, 2) verify whether a file is compliant with a given model, and 3) generate new PE files that satisfy the union or intersection of a set of models. The advantage of our approach with respect to previous efforts is that it can capture aspects of PE parsers in a comprehensive way, thus allowing us to exhaustively explore the search space in a structured way — something that other automated approaches (such as fuzzing) could not do.

### 3.4.1 Constraints Extraction

In our study, we model the checks performed on all the PE headers that are relevant to the loading process. These include the *MZ header*, the *COFF header*, the *OptionalHeader*, and the *Section Table*. We also modeled the entries in the *Data Directory* that play a role during loading, namely the

*Import Directory*, the *Import Address Table*, the *Loader Configuration Directory*, and the *Base Relocation Directory*.

Our first objective is to extract the list of checks that a program performs when it loads a PE file. Several static analysis techniques exist that may help to automate this process. For instance, both symbolic execution and taint analysis can keep track of all the operations a program makes on its inputs and translate them into formulas. However, these approaches have severe limitations when applied to complex software. For instance, the Windows loader often stores and manipulates information in linked lists. This makes the relationship between a byte in the original PE file and a byte "tested in memory" very complex to describe and it often results in over-tainting large parts of unrelated memory. Moreover, existing tools (even after a considerable manual effort) could not scale to a typical OS loader's complexity. In fact, OS loaders are tightly connected to other low-level components of the operating system (such as the memory management and the code responsible for populating all process data structures in the kernel), and, as a result, the inability to isolate the loader itself from the rest of the OS code often results in a constant path explosion and constitutes a major problem for symbolic execution and taint analysis engines.

While it might be possible to prepare the target code manually in a way that could be analyzed by automated techniques, the effort would need to be repeated for each application. We attempted to perform such a task during our research, but we realized that it was faster and more advantageous to manually extract the constraints than to reverse the code to understand which paths were safe to prune for the symbolic execution engine. Moreover, in the context of analyzing a small number of very complex real-world software, this manual process, when assisted with an automated regression testing environment, is significantly less error-prone than complex automatic approaches (e.g., symbolic execution) due to various shortcuts these approaches need to use when dealing with complexity.

We now provide more details about how this manual effort was conducted on the Windows loader, by far the most challenging software we modeled. This approach can be used to model other proprietary products with slight adjustments, while open-source software products — such as the other pieces of software we modeled — present fewer reverse-engineering challenges. We began by identifying the functions *MiCreateImageFileMap* in *ntoskrnl.exe* and *LdrpInitializeProcess* in *ntdll.dll*, which can be considered as the entry points for the kernel- and the user-

space phase of the loading process, respectively. Using *IDA Pro*, and assisted by the debug symbols provided by Microsoft, we decompiled these two functions as well as the other routines that they invoke. Most of these routines return either a success or an error code. Our manual analysis consisted of finding all these exit conditions, tracing back their dependency to the input file, and encoding them using our modeling language.

Once all the functions of interest have been modeled, we tested the model to find bugs or missing constraints. To do this, we generated a number of test cases and ran them while monitoring the operating system with a kernel-debugger (we will explain the technique for generating test cases in the next section). This allowed us to gain confidence in the accuracy of our models.

### 3.4.2 Modeling Language

We designed a custom language tailored for describing and encoding the knowledge acquired with our manual analysis effort. The rationale for this choice is that very few tools have been developed over the years for systematizing logic constraints like the ones we want to model. The few available options are not tailored for our purposes and relying on such tools would introduce a significant overhead for the analyst, making the modeling process more time-consuming. For example, the *SMT-LIB* language [BST+10] lacks support for modeling loops, while Dijkstra's Guarded Command Language [Dij75] does not support structured types, which proved to be extremely useful for modeling the Windows loaders.

Instead, our language is designed for the purpose of modeling compliance checks on the PE Format (or any other similar format, like ELF). Moreover, models written in our language can be automatically translated in *SMT* queries to ensure that the constraints of the model are coherent and to generate test cases that "satisfy" the constraints of the model. We discuss several possible use cases in the next section.

From a high-level perspective, each model is a list of statements. Our language supports various types of statements, each of which aims at capturing (and making it easy to capture) specific traits of the file format we want to describe. We now document the various statement types and features of our language. The reader can find a toy example of a model written in our language and an excerpt of one of the models of the Windows loader in Appendices A.1 and A.3, respectively. Moreover, the interested reader can review the full models at `https://github.com/eurecom-s3/loaders_modeling`.

**Input Definition.** The first type of statements allows us to create an *input symbol* of a given size in bytes that other statements can predicate on. Although our language supports multiple input definition statements, in our models we introduce only one symbolic input representing the *entire PE executable*. By doing so, we can treat the different components of the PE format as interconnected and interdependent entities, allowing our models to capture complex constraints involving fields of different headers.

**Symbol Definition.** The second type of statements allows us to introduce additional *symbols* (e.g., labels) and associate them with the result of operations on input or other symbols. The current version of our language supports arithmetic and bit-wise operations, as well as more complex operations commonly used to deal with the PE Format, e.g., *ALIGNUP* and *ALIGNDOWN* for respectively rounding up or down to a certain power of two.

**Predicates.** The third type of statements allows us to specify boolean comparisons between expressions of existing symbols, which evaluates to either *true* or *false*. In these statements, comparison and logic operators can be used, as well as more complex operators, e.g., *ISPOW2* to check if the operand is a power of two.

**Terminal Predicates.** Predicates in our language can be either *terminal* or *non-terminal*. A terminal predicate *must* be satisfied for an input file to be considered compliant with the model. Non-terminal predicates, instead, do not have such a requirement and can therefore be helpful when dealing with conditional predicates, which we discuss next.

**Conditional Predicates.** These allow encoding predicates of the form $P : \ A \Rightarrow B$, where $A$ is a non-terminal predicate. In other words, *if* the predicate $A$ is true, *then* the boolean predicate $B$ *must* also be true for the overall predicate $P$ to be satisfied.

**Structured Types.** As we mentioned in Section 3.2, the PE Format consists of many structures containing different fields. Our language implements a type system that makes models more readable by enabling to cast structured types on the original file and allowing the use of mnemonic names for each of the header fields. Types can be defined as C-like structures and can be used in all the statements discussed above.

**Loops.** The last construct supported by our language allows the encoding of *loops*. Loops are frequently used when parsing PE files, for instance to enforce that all entries in a list or array satisfy the same constraint. Once again, all the statements discussed above are supported within a loop.

## 3.5 Using Models

Once a human analyst has modeled the software constraints in our language, the models can then be automatically translated into a formal representation that can be used for various use cases. Moreover, we note that while the (manual) model extraction may theoretically contain imprecisions (we discuss in Section 3.6 how we experimentally validated them), all subsequent analysis steps have soundness guarantees.

### 3.5.1 Sample Validation

The first use case of our models is to determine whether a given PE executable meets the requirements of a specific loader. For this purpose, we rely on the *procedural interpretation* of the model. This technique consists of interpreting the statements in the model sequentially, applying each of them to the executable under analysis.

Symbols defined via *symbol definition* statements are assigned concrete values according to the input file and predicates are evaluated to their boolean values. In case any of the *terminal predicates* is *false*, the validation process stops, and the executable is marked as non-conforming to the model. Conditional predicates are also evaluated to concrete values, but in their case the sample is rejected only if their precondition is true as well. If all terminal predicates evaluate to true, and thus no constraint is violated in the process, the executable is marked as conforming to the model.

This use of our models can be relevant as part of the dynamic malware analysis pipelines that are employed by all major security companies that offer malware detection and analysis products. In particular, it could be useful as a reliable pre-filtering stage that either selects the appropriate sandbox (based on the OS that can run the sample) or quickly discards malformed binaries that would not run successfully anyway.

### 3.5.2 Sample Generation

In this second use case, we describe how we can automatically generate concrete samples compliant with a given model. This is possible because models written in our language can be transformed into SMT decision problems over the *BitVector* theory. The input of the problem is a symbolic *BitVector* (of fixed size) representing the executable file. Each type of statement in our language can then be translated in an appropriate SMT problem, as follows.

**Symbol Definition and Structured Types.** Each *symbol* defined in a model is transformed into a symbolic value that can be processed by an SMT solver. Moreover, for each operation that puts a new symbol in relation to existing symbols, our tool generates appropriate predicates that encode their relationship. Structured types are handled in a similar fashion, by using the appropriate offsets in the symbolic input to convert aliases into concrete predicates.

**Predicates.** Each predicate in our language is transformed to a boolean formula usable by an SMT solver. These SMT predicates are then used to create appropriate constraints, depending on whether they are terminal or non-terminal.

**Terminal Predicates.** Terminal predicates *must* be satisfied for an input file to be compliant with a given model. Our tool performs the logic conjunction of all the terminal predicates to create the SMT problem's predicate that it then feeds to the solver.

**Conditional Predicates.** Conditional predicates *must* be satisfied if their precondition evaluates to true. To model them compatibly with an SMT solver, for a conditional predicate of the form $C \Rightarrow D$, we create the following SMT constraint: $\neg C \vee D$.

**Handling Loops.** SMT theories do not have an equivalent construct to our loop blocks. To address this problem, we handle loops in our models by using *loop unrolling*. In other words, each statement in the loop block is executed up to a fixed amount of times (defined in the loop statement).

Finally, we create a single constraint that encodes the logical conjunction of all the constraints mentioned above (i.e., if $P_n$ is the n-th constraint in the model, we consider the constraint $\bigwedge_i P_i$). We then feed this single constraint to the SMT solver and ask it to produce a concrete input that satisfies all the predicates captured by our model. In the case of models that describe the compliance checks performed by a program on a PE executable, the SMT solutions are effectively PE headers that pass these checks. We use this observation to generate valid executables for the software we modeled.

**Generating PE Files with Valid Code.** A PE file generated by the SMT solver would pass all the checks but it would not execute any meaningful user-space code (because the loader does not check this part). However, this makes it harder to test whether a generated sample is valid and executes correctly. Therefore, we added a component to specify which code the generated PE file should execute and "plug" it into the generated files.

We (successfully) tested our system with two examples, the first executing a single "exit(0)" syscall and the second printing "Hello World!". For more details, Appendix A.2 shows a concrete example of the translation of a model into an SMT problem.

### 3.5.3 Corner Cases Generation

Our next use case focuses on generating a multitude of different test cases that *explore* the corner cases imposed by the PE loaders. To do so, we leverage *non-terminal predicates*, i.e., predicates that acts as preconditions for *conditional statements*. Since they do not need to be satisfied, we can consider them as *free variables* of the SMT problem.

Based on this observation, we automatically derive a number of SMT problems in the following way. Let $S$ be the predicate of the SMT problem, and $Q$ a non-terminal predicate of the model. Now consider the two following SMT problems: $S' \leftarrow S \wedge Q$ and $S'' \leftarrow S \wedge \neg Q$. By construction, even if $S'$ and $S''$ are different from $S$, their solutions (if they exist) necessarily satisfy the original problem $S$ too (because, by design, $S$ is less restrictive). However, these solutions may differ substantially. In fact, in the first, the free constraint is asserted, which means that the *conditional statements* that have $Q$ as precondition must be satisfied. The same is not true for the solutions to the second problem.

We can generalize this process to an arbitrary number $n$ of predicates used as conditions by building new problems that either assert or negate each combination of them—thus resulting in at most $2^n$ different generated samples. In practice, the actual number of generated test cases can be lower because there is no guarantee that some combinations of the *free constraints* are not incompatible with each other, thus making the corresponding SMT problem unsolvable.

### 3.5.4 Differential Analysis

Another use case of our work is to combine the models of two different software and generate samples that either satisfy both or only one of the two.

For example, we can select two SMT problems built from distinct models (for instance, two versions of Windows or a version of Windows and an antivirus), whose predicates are $S_1$ and $S_2$, respectively. Our system can then generate a third SMT model with the following predicate: $S_{diff} : S_1 \wedge \neg S_2$. If this SMT problem is satisfiable, its solutions are samples that

pass all the compliance checks of the first software but do not satisfy at least one of the constraints in the second model. On the other hand, under the assumption that the models correctly reflect the behavior of the software, if the problem is not satisfiable, it *guarantees* that no such samples exist.

We can also use this technique to prove that the two models are *equivalent*. In fact, if we cannot generate any sample that satisfies $S_1$ but not $S_2$ and neither any input that satisfies $S_2$ but not $S_1$, we can conclude that the two models are enforcing the same constraints.

### 3.5.5 Differences Enumeration

In our final use case, we can take the *Differential Analysis* technique one step further to find the *exact* constraints in the two models that make them behave differently.

The idea is to use an iterative process that starts with the differential analysis between the two models to compare. If the differential analysis problem has no solution, the process terminates. If, instead, a solution is found, we identify the predicates in the negated model that were false in the produced test case. We then add these predicates to the set of constraints negated at least once (from now on, $N$).

The process then continues with solving SMT problems of this form:

$$S^n_{diff} \leftarrow S_1 \wedge \neg S_2 \wedge S_{support}$$

in which $S_{support}$ represents the logic conjunction between a subset of the predicates in $N$. By construction, the solutions of this SMT problem (if any) meet the constraints in $S_{support}$ but violate at least one constraint in $S_2$. We then add the new violated constraints to $N$ before repeating the process until we have used all the subsets of $N$ (including $N$ itself) in one iteration.

Note that this approach does not guarantee that *all* the differences between the two models are found when the process ends. To have this guarantee, one should build the $S_{support}$ terms as the logic conjunctions of all the subsets of the predicates in $S_2$. However, this requires a number of iterations that grows exponentially in the number of predicates in $S_2$, limiting its applications in practice.

On the other hand, we believe that our approach represents a good trade-off between scalability and precision. In fact, the number of iterations required is lower since it is exponential only in the number of the

negated predicates. Moreover, the models that we compare are substantially similar, meaning that only a few of the constraints are not coherent between the models.

### 3.5.6 Implementation

We implemented a model analysis framework (available at https://github.com/eurecom-s3/loaders_modeling) that can perform all the operations described above. The tool consists of three different components. The first is the *Language Front End* responsible for parsing the input models and lifting them into an intermediate representation in the form of an abstract syntax tree. This component also handles the typing system by resolving the mnemonic fields of the structured types into offsets in the symbolic input.

The second part is a *Python Backend*, which performs the *sample validation* task by sequentially validating each statement in the intermediate representation on a sample provided as input.

Finally, the core of the framework is the *Z3 Backend* that implements the logic to translate models into SMT problems and solve these problems by using the z3 SMT solver [DMB08] to generate samples. It also provides an interface to combine an arbitrary number of models together and to perform *Corner Cases Generation*, *Differential Analysis*, and *Differences Enumeration*.

## 3.6 Models Evaluation

For our study, we modeled the loading process of three versions of Windows: Windows XP SP 3, Windows 7 SP 1, and Windows 10 (v. 1909). On average, each model contains 269 statements, of which 78 are terminal predicates.

Since all constraints were extracted by manual analysis, which may be affected by imprecisions, we evaluated how tightly these models capture the behavior of the target software components. To this end, we conducted two sets of experiments to assess whether the models are under-constrained (i.e., "too loose" in accepting invalid files) or over-constrained (i.e., "too strict" in rejecting valid files).

### 3.6.1 Assessing Under-Constrainedness

The objective of the first experiment is to verify that our models are not under-constrained with respect to the real OS. That is, if our model says "this file is a valid executable for a given OS," then the OS should be able to load and execute that file. Our approach consists of generating samples "at the boundary" of our models and attempting to execute them in the real OS: if a valid file according to our model does not run in the OS, our manual analysis missed important constraints (i.e., the model is under-constrained).

To generate these "extreme" samples, we used the *Corner Cases Generation* technique introduced in Section 3.5.3. By construction, all these samples are guaranteed to be valid according to the model. Moreover, since this technique recursively explores all the relevant free variables in the model, the generated samples can be seen as representatives of all models' corner cases. In details we generated 80 test cases from the model of Windows XP, 72 for Windows 7, and 20 for Windows 10. Each sample has been generated by combining the operating system model with the model responsible for adding the code of an "exit(0)" syscall at the entry point. This allowed us to infer whether the sample ran correctly by observing its return code (*%ERRORLEVEL%* according to the Windows terminology).

All the test cases generated by each model ran successfully under the respective operating system.

### 3.6.2 Assessing Over-Constrainedness

The goal of the second experiment is to verify that our models are not over-constrained compared to the real OS, i.e., that our models do not include erroneous constraints that were not present in the loader code. In other words, if the OS can load a given file, then the corresponding model should output that "this file is valid." To this end, we first built a dataset of 2543 real-world PE executables that we collected from the community-driven repository of the Chocolatey [cho] Windows package manager. Note that, although these samples are very likely valid PE executables, they would not necessarily run under *all* the three Windows versions under analysis. We processed each file with the *Sample Validation* technique (discussed in Section 3.5.1) and we identified the samples that our models flagged as invalid: given the nature of the dataset we start with, these samples represent potential imprecision of the models. Lastly, we verified whether these invalid files would actually run under the corresponding OS.

Out of the 2543 samples, our models predicted that 632, 261, and 86

were invalid according to our models of Windows XP, Windows 7, and Windows 10, respectively. To verify the accuracy of the prediction, we then executed each of these samples in a VM running the respective version of Windows OS: *all samples failed to run, precisely as predicted by our models*. We believe this is strong evidence that our models are not over-constrained. We also investigated the reasons behind this high rate of invalid PE samples. The most frequent problem is that these are PE files targeting more recent Windows versions using the *SubsystemVersion* fields. Other samples are invalid because they target other architectures than Intel x86, or because they are kernel modules that cannot run as standalone executables.

As an additional experiment to check that the robustness of our execution setup, we also ran all the "valid" samples and verified that they were all loaded properly in the respective OS. Note, however, that some of them could not be properly executed due to missing DLL dependencies, which is a problem that is not related to whether a given sample is compliant with the PE format.

## 3.7 Differential Analysis

In this section we present the results of the differential analyses we performed on our models. In particular, we present the discrepancies we found between the versions of the Windows loader we analyzed, as well as those between each Windows loader and three open-source tools commonly used in malware analysis, namely the ClamAV antivirus [Cisa], the yara signature engine [yara], and the reverse-engineering framework radare2 [rad].

For the three versions of the Windows loaders and ClamAV, we compare the models of their compliance checks and their memory mappings. On the other hand, we compare only the models of the memory mapping operations performed by yara and radare2. This asymmetry is due to the latter two not performing any compliance checks when analyzing PE executables.

The analyses were performed by using the *Differences Enumeration* technique presented in Section 3.5.5. For the sake of completeness, each discrepancy we report has been manually validated by feeding the corresponding software with the test cases that the differential analysis generated.

Table 3.1: Source of discrepancies and in which differential analysis they were found

| | Discrepancies | | | | |
|---|---|---|---|---|---|
| | **W1** | **W2** | **W3** | **W4** | **W5** |
| **XP vs 7** | ✓ | ✓ | | | ✓ |
| **XP vs 10** | | ✓ | ✓ | ✓ | |
| **7 vs XP** | | | | | |
| **7 vs 10** | | | ✓ | ✓ | |
| **10 vs XP** | | | | | |
| **10 vs 7** | ✓ | | | | ✓ |

### 3.7.1 Discrepancies among Versions of the Windows Loader

Interestingly, we found differences among all these three versions of the Windows loaders. For what concerns compliance checks, we found a number of discrepancies, discussed next. We note these represent *the exhaustive list* of discrepancies between the models we created for this work and that all these were found automatically.

**[W1] ImageBase = 0.** Windows 7 considers as invalid any executable with a value of *ImageBase* equal to 0. We did not find the same behavior in either Windows XP or Windows 10.

**[W2] SizeOfHeaders $\geq$ offset(COFFHeader) + 0x5d.** For executables with *SectionAlignment* greater than the page size, both Windows 7 and Windows 10 expect the *SizeOfHeaders* to be greater than the offset in the file of the *COFFHeader* plus a constant. Windows XP, on the other hand, does not enforce this constraint.

**[W3] Relocations for other architectures.** Both Windows XP and Windows 7 handle architecture-specific relocation entries, even if they are not "meaningful" for the running system. On the other hand, Windows 10 only allows generic and Intel x86-specific relocations.

**[W4] AddressOfEntryPoint < SizeOfHeaders.** If the *AddressOfEntryPoint* of an executable lies within the first *SizeOfHeaders* bytes of the image, Windows 7 and Windows 10 discard the executable as invalid. The same does not happen under Windows XP.

**[W5] SizeOfImage > endof(SectionTable).** Windows 7 is the only version that does not load executables in which the offset of the last byte of the *SectionTable* in the file is greater than the value of *SizeOfImage*.

Table 3.1 provides a summary of the discrepancies found during the differential analyses. In particular, the ✓symbols in the table indicate whether a discrepancy was found during a differential analysis of two versions of the loader. For example, the ✓in first row (XP vs. 7), first column (W1) indicates that the first discrepancy (*ImageBase = 0*) was found while generating samples that comply with the model of Windows XP, that did not satisfy the model of Windows 7.

It is interesting to note how our analysis did not find any discrepancy based on the *OperatingSystemVersion* fields. In fact, according to the specifications, these fields should indicate the "version of the operating system" required to run the executable. However, in our investigation, we did not find evidence of any check performed on these fields, which can, therefore, assume any value. This is an example of the significant differences between the specifications of the PE Format and its software implementations (which are both maintained by Microsoft). On the other hand, the Windows loader checks that the *(Major|Minor)SubsytemVersion* fields are within a range that varies across the three versions of the loader we analyzed. Certain values of these fields also enable some version-specific features, such as the *Control Flow Guard*[Mic18a] extensions of the *Load Config Directory*, which the Windows 10 loader validates if the *SubsystemVersion* is greater than "6.3." Version-specific features represent one more cause of discrepancies in the behavior of the Windows loaders.

Our differential analysis on the memory mapping operations also highlighted a difference in how Windows 7 and Windows 10 map PE executables compared to Windows XP. The difference affects the first region of the memory map, the one that contains the PE headers. Each version of Windows maps the PE headers in the process address space. However, if the *Section Alignment* of the executable is greater than the page size, Windows 7 and Windows 10 copy in this region only up to *SizeOfHeaders* bytes, while Windows XP copies up to 4KB from the executable file.

### 3.7.2 Compliance Checks Analysis of ClamAV

We now document the results of a differential analysis to produce test cases that comply with the constraints enforced by the operating system loaders, but violate those of the antivirus. For this experiment, we focus on ClamAV. The net result of each of our findings is that while these executables would run under the different Windows operating systems without problems, ClamAV would not be able to consider them as "fully valid PE files," and it would not be able to use signatures that rely on specifics of

the PE format [Cisc, Cisb].

It is also interesting to note how the differences reported by these differential analyses are completely different from those reported between the different versions of the OS loaders—showing once more how each developer interpreted in her own way the file specification. We now discuss the discrepancies we identified.

**[C1] SizeOfOptionaHeader.** As we explained in Section 3.2, this field is used to determine the length in bytes of the *OptionalHeader*. ClamAV expects its value to be greater or equal to the size of the structure defined by the PE Format. None of the Windows loaders we analyzed in our experiments enforce this constraint.

**[C2] NumberOfSections.** ClamAV expects at most 96 entries in the *SectionTable*. To the best of our knowledge this was the maximum number of sections in an executable supported by older versions of the Windows loader, while none of the versions of Windows we analyzed still enforces this constraint.

**[C3] Section Virtual Address.** For executables with *SectionAlignment* less than the page size of the architecture, the Windows loaders *do not* check that the starting addresses of the sections are multiple of this value. ClamAV, on the other hand, performs this check regardless of the value of the *SectionAlignment*.

### 3.7.3 Memory Mapping Analysis of ClamAV, radare2, and yara

We modeled the memory mapping operations of three popular software that deal with the PE format, namely ClamAV, radare2 [rad], and yara [yara]. Each of these software needs to provide some sort of memory mapping of PE executables. For ClamAV and yara, this memory mapping can be used to write malware signatures that rely on the content of the memory image. Radare2, on the other hand, provides the user a full-fledged memory representation of the PE executable to support her analysis and reverse engineering.

Under the hood, memory mapping operations are usually implemented by means of a primitive that maps *RVAs* in the memory image to offsets in the original file. We therefore extracted and modeled the implementation of this primitive in all the three software and performed a differential analysis between them and the three versions of Windows. Our system found discrepancies in the memory mapping operations of all the

three software. In particular, we discovered that the nature of these differences was the same and it was due to incorrectly mapping PE executables that have a *SectionAlignment* lower than the page size. For these executables, all three versions of the Windows loader we examined copy the entire file *as is* in memory, starting at *ImageBase*, regardless of the entries in the section table (thus resulting in *RVAs* to be equivalent to file offsets). On the other hand, the three tools always rely on the section table to convert *RVAs* into file offsets.

Our differential analysis automatically produced test cases that leverage this key difference, by exploiting the fact that PE executables with a low value of *SectionAlignment* do not need a section table at all. Once again, the fact that all these three software make the same mistake shows that the PE specifications do not provide a clear and comprehensive set of guidelines for handling PE executables, nor they describe accurately the actual implementation of the Windows loader.

## 3.8 Bypassing Popular Analysis Tools

The discrepancies we discussed in the previous section have significant repercussions in the field of malware detection and analysis, as they undermine the trust we have in popular tools. This section discusses several examples of real executables that are not correctly supported by popular tools. It then presents the results of our investigations to determine 1) whether these discrepancies can be found among benign samples, and 2) whether there is evidence of malware samples that already exploit them as part of their attacks in-the-wild.

**ClamAV.** We modified the headers of real PE files to deceive ClamAV into not considering them as fully valid PE executables and bypassing signatures relying on specifics of the PE format. For instance, by simply adding empty sections at the end of the section table and updating the *NumberOfSections* field accordingly, the malware would still be valid for the Windows loader, but ClamAV would have issues using some PE-specific signatures. With minor adjustments, malware authors could also easily abuse the other discrepancies by, for example, lowering the value of *SectionAlignment* below the value of the page size, or by modifying the virtual address of one of the entries in the section table, so that the new value is not a multiple of the value of *SectionAlignment*. We created several proofs-of-concept that showcase these problems, and none of these attacks altered in any way the malware behavior.

**Yara.** We crafted a PE file that evades two key features commonly used in malware signatures: The resolution of imports and the pattern-matching applied at specific virtual addresses (implemented with the keyword *at*). The technique we used leverages the discrepancy in the memory mapping operation described in the previous section, and mainly revolves around creating a PE executable with a low value of *SectionAlignment*. To deceive the imports resolution, we placed the name of the imported DLL outside the memory ranges covered by the section table. The same can be done to mislead actual signatures that use the *at* keyword. For our example, we placed the entry point at an *RVA* that is not covered by the section table, but one could hide *any* portion of the code with the same technique.

**Radare2.** Reverse-engineering tools are not immune to mishandling the PE format either, and our experiments show that they can be easily exploited to confuse both manual and automatic analysis based on these tools. As a proof-of-concept, we developed a technique to selectively hide portions of code from the radare2 framework. This was achieved by lowering the value of *SectionAlignment* and then adjusting the entries of the section table to minimize the number of bytes that radare2 maps in memory (by reducing the value of *SizeOfRawData* field in each entry of the section table).

**Dynamic analysis pipelines.** The showcased examples aim to evade static malware analysis, but dynamic malware analysis techniques can be evaded too. In fact, dynamic analysis pipelines rely on an instrumented version of an operating system to carry out their job. As we saw in the previous section, discrepancies among versions of the Windows loader exist, and no version can load *all* the programs that the others can. Most dynamic malware analysis pipelines run each sample only with one Windows version, which is often outdated and different than what end-users choose[YIT+16]. Without a way to statically identify the OS version to use, it is difficult to ensure that every file is correctly analyzed in the sandbox.

**Measuring prevalence among goodware.** Intuitively, one would not expect to find header "malformations" in benign software. In the vast majority of the cases, in fact, goodware is compiled by using mature and well-tested toolchains that are unlikely to produce headers that trigger different behaviors in different parsers. To test the validity of this assumption, we analyzed the dataset of goodware we used in Section 3.6.2, searching for samples that exhibit any of the causes discrepancies highlighted in Section 3.7. Out of the 2543 samples in the dataset, we found only three samples whose headers showcase any of the malformations. All three of them

are resource-only images (i.e., PE files that do not contain any code, but only data) and had their *AddressOfEntryPoint* fields set to 0, matching the conditions of the *W4* discrepancy. According to the Windows APIs documentation [Mic18b], resource-only images undergo a different loading process, which does not perform many of the operations that the regular process does, ultimately enforcing fewer constraints on the PE headers. Thus, there is no evidence of prevalence among goodware of conditions that can lead to discrepancies in the PE loading process. In other words, in none of the samples in our dataset, could we find PE headers that would trigger any of the discrepancies reported in this chapter. This suggests that, if such peculiar headers were to be found, they most likely would have not been produced accidentally.

**Evidence of usage in-the-wild.** At last, we wanted to investigate whether real-world malware is already exploiting the discrepancies we found. To this end, we run a LiveHunt campaign on VirusTotal [vir] from Oct. 7, 2020, to Oct. 19, 2020. The LiveHunt service allows researchers to scan all the samples received by the VT platform with custom signatures, written in yara, in real-time. For our campaign, we wrote yara rules that match the discrepancies we found in the compliance checks reported in the last section, with the only exception of the one concerning architecture-specific relocations.[1] We also wrote one additional rule that matches samples that exhibit a value of *SectionAlignment* lower than the page size. As we saw throughout the chapter, this is the precondition for all the discrepancies in the memory mapping operations.

Our LiveHunt identified a total of 467 samples. Table 3.2 presents a breakdown of the samples grouped by which yara rule matched and by the number of AV detections. 73% of the samples were marked as malicious by 20 AVs or more, with an average of 36.6 AV detections (out of 74) per sample.

Despite evidence that the presence of discrepancies is rare in benign samples (as discussed earlier), it is challenging to determine that, with certainty, these LiveHunt samples are intentionally abusing them. Nonetheless, there are some cases for which there is indeed relevant evidence. For example, we found 77 samples with more than 96 sections, which would interfere with ClamAV's scanning process. Intrigued by the (relatively) high number of encounters, we manually analyzed some of these malware

[1] The PE module of yara does not provide APIs to access the relocation table, and parsing it using the yara language is not possible as it lacks support for a loop construct in which the number of iterations is not determined before the loop starts, which is necessary to parse the table.

Table 3.2: #Samples reported for each discrepancy

| | | Windows Loaders | | | | Windows vs. ClamAV | | |
|---|---|---|---|---|---|---|---|---|
| | **Align** | **W1** | **W2** | **W4** | **W5** | **C1** | **C2** | **C3** |
| **# Samples** | 301 | 37 | 43 | 27 | 1 | 15 | 77 | 1 |
| $[0, 5)$ **Detect.** | 59 | 14 | 3 | 15 | 0 | 0 | 1 | 0 |
| $[5, 10)$ **Detect.** | 36 | 1 | 3 | 0 | 0 | 0 | 1 | 0 |
| $[10, 20)$ **Detect.** | 5 | 4 | 2 | 1 | 0 | 1 | 0 | 0 |
| $\geq$ **20 Detect.** | 201 | 18 | 35 | 11 | 1 | 14 | 75 | 1 |

samples and found that they often include *exactly* 97 sections: this is unlikely a coincidence as this is the minimum number required to trigger the constraint in ClamAV. The relatively large number of samples using this trick suggests, in our opinion, that malware authors are actively employing this as an evasion technique. However, we do not know whether these tricks are used to specifically target ClamAV or whether they are additionally targeting other antiviruses, which may be affected by similar problems.

As another noteworthy example, VirusTotal reported one sample exhibiting the discrepancy involving the end of the section table (i.e., rule W5). It appears that this malware purposely crafted it to escape static malware analysis. In fact, on top of its header's peculiar structure, the sample also showcases several anti-analysis techniques, including anti-disassembly tricks and runtime library loading.

Last, most of the identified samples exhibit a value of section alignment lower than the page size. We believe it is likely that the samples' authors adjusted this value on purpose. In fact, the default value of section alignment for all the Windows toolchains we tested is precisely the page size, which is confirmed by very few encounters in regular PE executables.

To properly understand the results presented so far, it is important to contextualize these numbers with respect to the big picture. In fact, on an average week, VirusTotal processes around 3 million submissions of Windows executables [Vir21], which suggests that the exploitation of the discrepancies is a relatively niche phenomenon at the moment. However, we believe that evidence of in-the-wild use of these discrepancies is concerning. To the best of our knowledge, we are the first to report these discrepancies publicly, and we hope our work will help the community deal with this problem in a timely manner.

## 3.9 Discussion

The results presented in this chapter prove that different software handle the PE format in different fashions, leading to incongruities in both *compliance checks* and *memory mappings*. Our models show that there are as many ways to interpret a PE file as there are versions of Windows. In Section 3.8 we showed how these discrepancies allows attackers to bypass popular analysis tools and pipelines, and to create targeted executables that would run on a certain version but would be discarded as malformed by the loaders of other versions. We reported the bugs we identified to the developers of the tools; ClamAV developers have already acknowledged the problem and they are working on fixes. We believe, however, that it would be significantly more complicate to "properly" address the various aspects discussed in this chapter.

**There is not a single reference (or even a correct) implementation.** We believe that security tools should allow the user to decide which model to use on a case-by-case basis. We note that this goes beyond the relatively well-known "the analyst should be able to select which type of Virtual Machine / Windows version to use for dynamic analysis " as the problem affects all static reversing tools as well. As we have shown in this chapter, popular reversing tools can be completely bypassed and they can be fooled to return misleading and/or inaccurate data to the analysis client (e.g., a human analyst or a processing block in an automated pipeline)

So far, *no static reversing tool (including the popular commercial options) has even the "concept" of letting the user selecting which loader to emulate.* Over the last decade, instead, they have all tried a best-effort approach to implement a *single* loading process that attempts to be flexible and catch the various differences. This work wants to sound the alarm bell that this long-time effort is bound to failure as *there is not a single "correct" implementation*, but there are as many as the OS versions. Other than fixing bugs to patch the discrepancies, we argue that the correct approach to eradicate this problem is to continue our effort to document the parsing models adopted by different software implementations and encode this knowledge in formal models that can be included in other tools and selected by the user.

**On the need of formal specifications and reference implementations.** Until now, developers of security tools had to implement their own PE parsers as the constraints and operations performed by the Windows loaders were not documented. At first glance, parsing the PE Format may seem a simple task, on top of which a large body of research and com-

mercial tools are built. As our experiments show, however, there are many corner cases that are not sufficiently described in the PE specification, and different tools and operating system handle them in very different ways. This is a systemic issue that represents a concrete threat especially in adversarial fields like malware analysis, in which every wrong assumption may open up new avenues for evading detection mechanisms.

We argue that such problem should be tackled at its roots. One of these is the lack of a formal specification of the format that defines what a well-formed PE executable looks like. The ambiguities in the current specifications have ultimately lead to incongruities in how PE executables are handled. A possible solution to this problem could be to systematize the PE Format by means of formal methods. This would not only ease the development of new tools and loaders, but it would also provide reliable ways to discover discrepancies in their implementations.

One other way to avoid discrepancies among the large number of software dealing with the PE format could be to have a well-documented, publicly available reference implementation of the Windows loader, to which the new versions of the loader comply. This would ease the task of writing new reverse engineering tools and antiviruses, since they would not require developers to guess or to manually reverse engineer the Windows operating system internals.

While we believe this to be the best solution for the future, our work provides a solution to deal with both past and present versions of MS Windows. All our models and the tools required to use them for validation, test case generation, and comparison among tools are available at `https://github.com/eurecom-s3/loaders_modeling`. Thanks to our effort, PE parsing libraries and tools (such as pefile [ero] and pev [pev]) can provide different options to their users for choosing to interpret and/or validate a file according to one model or another.

## 3.10 Related Work

Program loaders are core components of an operating system and, as such, have been widely studied by the research community. We therefore organize the discussion of previous research in this field along four categories.

**Edge cases.** Researchers have shown the limitations of static analysis approaches in parsing binary formats like ELF [J. 13] and PE [ska06, roy], by abusing relocations to hide malicious code from static analysis tools.

Ge et al. [GPJ17] used relocations to alter the memory permissions with severe security implications. Other researchers showed how a single byte can break several ELF parsers and still execute on the target machine [ule19], and how to use ELF metadata to backdoor a `setuid` application [SBS13]. For what concerns the PE ecosystem in particular, Albertini [A. 13] created PE executables that executes different instruction on different Windows versions, by leveraging discrepancies in the loaders implementations of these operating systems. Albertini also developed a collection of proof-of-concept binaries that showcase exotic features of the PE file format [A. ]. Previous works, like those by Vuksan et al. [VP11] and Huang [Hua06], acknowledged that PE specifications leave space for implementation choices and documented a series of "malformed" (yet valid) PE header layouts. All these works only scratched the surface of the problem of discrepancies in the PE ecosystem. In fact, all studies discussed only anecdotal examples and the authors never attempted to generalize nor to propose a comprehensive approach to eradicate the problem.

**Differential Parsing.** The problem of differential parsing arises when different implementations of parsers for the same format produce discordant results when fed with the same input. Researchers presented several attacks based on differential parsing. For instance, Kaminsky et al. [KPS10] demonstrated an attack against the X.509 infrastructure. Other attacks allowed privilege escalation on mobile systems, like the infamous Android "master key" attack [sau13] or the more recent iOS 0day for the plist parsers [Sig20]. Bratus et al. showed how to create a file that the Linux kernel- and user-space loaders parse differently [BB13]. These works show how researchers found inconsistencies among implementations of parsers of the same format. However, all these cases had been discovered manually without any guarantee of completeness. In this thesis we instead propose a framework to automatically find all the incongruities among two tools. While we only presented our techniques applied to the PE ecosystem, we also created models for software parsing the ELF format and we are confident that our approach can be expanded to other formats as well.

**Evasion.** Several studies have focused on detecting malware evasive behaviors [LKC11, KVK14, XZGL14] or have attempted to measure their prevalence, such as in the case of stalling code [KKK11, B. 15]. The work presented in this chapters fits in the line of research that identified in the mishandling of executable file formats a major avenue for malware analysis evasion. Previous works have presented single instances of this problem. For instance, Petsios et al. [T. 17] showed two cases in which ClamAV

failed to parse malicious ELF files that properly run on the operating system. Similarly, Kim et al. [KKD17] showed that many AV engines do not scan signed PE files and do not accurately validate the Authenticode signature that can be copied from benign applications. Again, this shows how evasion attacks are possible when AV engines handle PE files differently than the OS loader.

**Automatic Modeling of Protocol Stacks.** Brumley et al [BCL+07] presented a taint tracking-based automatic approach to create SMT models of the behavior of web servers handling HTTP requests. They used these models for differential analysis with a technique similar to the one we presented in our work. However, due to the intrinsic limitations of taint tracking, the models they produced cannot be complete.

# Chapter 4

# Beyond API Tracing

***Implementing a Generic and Practical Bypass Technique and Investigating the Semantic Gap between APIs and Syscalls in Windows***

## 4.1 Introduction

To counter the spread of malware, the computer security community put in place mechanisms to discover emerging threats and study their modus operandi, in order to recognize and stop them from causing harm. Behavior analysis holds a crucial role in providing a prompt and targeted response to new malware strains: Discovering what a program does is the first step towards determining whether its intents are harmless or malicious.

In the field of malware analysis, one popular way to encode a sample's behavior consists in gathering information about the system-provided functionalities (APIs) that it accesses. Both manual and automatic techniques collect and process such information, looking for evidence of malice in unknown samples. For a human analyst, the mere fact that a sample imports specific Windows APIs may represent an early red flag for malevolent intents. Anti-malware tools of all kinds (including sandboxes, antiviruses, and endpoint protection) log the APIs that programs execute to assess their maliciousness. Similarly, previous research works heavily depend on such API information for malware characterization and detection [KKB+06, SM07, PHL+15, KKK15, HBZ18, RT20].

Given its widespread adoption by the security industry, it is paramount

that the tools through which we collect API-level information provide genuine and trustworthy data. As discussed in this work, however, the techniques in use today have technical flaws that limit their reliability, opening the door to evasion mechanisms that malware authors soon added to their toolboxes.

A more fundamental problem affecting API-level behavior analysis, arises from the very premise of this approach. The (often overlooked) assumption underlying the adoption of the APIs for behavior encoding, is that programs (including malware) do indeed make extensive use of these functionalities. At first glance, this assumption seems completely reasonable, at least from a cost-benefit point of view. In fact, using the system-provided APIs allows programs to delegate complex tasks to the operating system without reinventing the wheel, thus optimizing implementation costs. Moreover, since APIs tend to be relatively stable, they also guarantee portability among different operating system versions.

Although this reasoning holds in most cases (and it surely does for the average non-malicious program) malware has different incentives, the first of which is hindering malware analysis. Successfully evading detection results in slower responses by the anti-malware community, thus extending the time frames in which the malware runs and monetizes uncontested. To put it in another way, from a malware author's perspective, being able to resist behavior analysis may justify investing more in the development process, even at the expenses of portability.

Over the last years, the security industry has witnessed an increasing trend of malware samples that avoid using APIs in favor of directly invoking the syscalls, the lowest-level interface between a program and the operating system on top of which the APIs are implemented. Luckily, for the moment, researchers have only found evidence of malware samples adopting this strategy for the early stages of their execution, often just to inject malicious code into a legitimate (hence, not monitored) system process. To date, writing entire malicious programs solely using syscalls is still a very challenging and error-prone process on Windows, and we speculate this is the reason for which we are yet to see fully API-less malware.

As a first contribution for this work, we introduce a novel approach that simplifies developing complex software that bypasses API-monitoring. Specifically, our approach allows one to write (possibly malicious) code employing high-level Windows APIs, and to automatically generate self-contained executables that do not need to interact with any OS module other than the operating system itself. The key idea behind our API-tracing-resistant program is to embed a self-contained *custom runtime*

*environment* (derived by the Windows libraries), which replaces the one provided by the operating system: Since the software uses its *custom runtime*, malware analysis solutions cannot capture any API information from its execution. Although simple in concept, the process of building self-contained Windows binaries presents a variety of hard technical challenges, that, as we will explain in detail, have their roots in the Windows operating system design.

We hope this contribution will be seen as a wake-up call for the academic and industry security community, since the adoption of these (and similar) techniques may make API-based behavior analysis significantly less effective. The syscall layer remains, in fact, the sole reliable bastion from which to monitor the behavior of an application. While APIs convey high-level and easily intelligible information, however, a syscall trace offers low-level information that requires a certain familiarity with the operating system internals to be fully understood. To make things worse, on Windows, the syscall layer appears to expose much less semantics than their counterpart on Linux. The `HttpSendRequest` API provides a staggering example of this semantic gap. Listing 4.1 shows the syscalls that a program invokes when using this seemingly simple API (for space concern, we only reported a few of the over *four thousand* syscalls that we recorded for this specific API). At first sight, this syscall list seems completely unrelated to the main functionality of the API. The trace, in fact, contains thread synchronization and inter-process communication primitives, interactions with the filesystem and the system registry, but nothing that we can intuitively trace back to a network operation.

Anecdotal examples such as the one presented above motivated our second contribution for this work: to measure and quantify the complexity of recovering API-level information from syscall-level traces on Windows. To this end, we run a large-scale dynamic analysis campaign, for which we executed over 23 thousand programs, collecting both the APIs and the syscalls they invoke. The results of this exploration are worrisome: We show that reconstructing API-level information from a syscall trace is profoundly complex and ambiguous even under strong, favorable assumptions.

In the spirit of open science, to allow the security community to replicate our work and further investigate the topic, we will open-source every relevant software artifact, dataset, and collected data.

**Paper structure.** The paper is organized as follows. Section 4.2 introduces common terminology and fundamental concepts about the Windows operating system needed to understand the technical details explained in

```
...
NtOpenEvent
ZwWaitForSingleObject
NtClose
ZwAlpcSendWaitReceivePort
ZwAlpcSendWaitReceivePort
ZwAlpcSendWaitReceivePort
ZwAlpcSendWaitReceivePort
ZwAlpcSendWaitReceivePort
ZwAlpcSendWaitReceivePort
NtOpenKeyEx
ZwQueryValueKey
NtClose
NtDuplicateObject
NtOpenThreadToken
...
```

Listing 4.1: Excerpt of the syscall trace of HttpSendRequest

the paper. Section 4.3 provides an overview of the state of the art of API-level monitoring solutions, with an emphasis on known bypass strategies. In Section 4.4, we present our novel approach to generate self-contained, API-monitoring-resistant executables, detailing the technical challenges and our solutions. Section 4.5 describes our large-scale dynamic analysis campaign and our attempts to quantify the semantic gap between APIs and syscalls. Section 4.6 discusses the limitations and implications of our work. Section 4.7 gives an overview of previous work similar or related to ours.

## 4.2 Background

This section provides the relevant background needed to understand the rest of the paper. In particular, it introduces the terminology and the fundamentals of the programming models of the Windows operating system, focusing primarily on the concepts of *WinAPIs* and *NativeAPIs*. We will then present the dynamically-linked libraries, i.e., the binaries that contain the implementation of the APIs, with a particular focus on three libraries that constitute the cornerstone of the Windows user-space runtime. We dedicated the last part to presenting *API Sets*, a concept intro-

duced in recent versions of Windows, and how their design influenced the already complex Windows library ecosystem.

### 4.2.1 The Windows Programming Model

Like any modern general-purpose operating system, Windows provides an *application programming interface*, a set of built-in functionalities provided by the operating system that would otherwise be very tedious, error-prone, or even impossible to implement in the context of a user-space program.

From a technical perspective, we can divide the Windows *API* [1] into two broad categories: *WinAPIs* and *NativeAPIs*. The first represents the higher-level layer, providing complex functionalities while hiding implementation details to the user program. Examples of *WinAPI* are those functionalities related to networking (including network protocol implementations for standard protocols and cryptography), the graphic user interface, and system services management.

*NativeAPI*, instead, is the lightweight and lower-level layer upon which the *WinAPI* relies. With few exceptions, most notably the C runtime library and the user-space loader, the actual implementation of the majority of the *NativeAPIs* resides in the kernel, which exposes them as system routines that the user-space program can access using special CPU instructions (e.g., `int`, `sysenter`, and `syscall` in the Intel x86/x86-64 ISA). In the remainder of the paper, we will indicate this subset of *NativeAPIs* with the generic term *syscalls*. Lower-level modules often depend solely on the *NativeAPI* layer. Such is the case of device drivers (that need to interact with the kernel directly) and those OS components that participate in the early initialization phases of the system startup, i.e., a moment in which the *WinAPIs* are not available yet.

The degree of their complexity, however, is not the only difference between *Native* and *WinAPI*. Indeed, while the latter is guaranteed to be stable among different major releases of the Windows operating system, the latter may change drastically. Introductions and removals of *NativeAPIs* happened various times throughout the years, as documented by [j00], which also showed that even the application binary interface for the same syscall (in particular, the so-called "syscall-number") changed among different releases of the same major version of the operating system. Moreover, while *WinAPIs* are thoroughly documented, some *NativeAPIs* are

[1] In this paper, we consider the *API* acronym as countable, indicating any of the functions that provided by the operating system. The same applies to *WinAPI* and *NativeAPI*.

not, as they are intended for internal use only.

Although, conceptually, it would be possible to write any program using either the *WinAPI* layer or the *NativeAPI* one (after all, the former builds on top of the latter), the choice is straightforward from a programmer's perspective: By providing better portability with less implementation complexity, the *WinAPI* layer is inevitably the most cost-effective in the vast majority of circumstances.

### 4.2.2 Dynamically-Linked Libraries

Unlike UNIX-like systems, in Windows, fully statically linked executables do not exist for two reasons. In the first place, both the *Native* and the *WinAPI* are version dependent, meaning that a fully statically linked binary would likely be compatible only with one version of the operating system. Furthermore, Microsoft provides no statically-linked versions of the system libraries, effectively preventing users from creating statically-linked binaries in the first place.

The lowest stable interface with the operating system is provided by the dynamically-linked libraries (from now on *DLLs*), which hide the version-specific implementation details of the APIs. To access the functionalities that the operating system provides, a process needs to load in its address space the DLLs that implement them. This operation can happen either during the initialization phase of the process, when the user-space loader of the operating system parses the *Import Table* in the *PE header* [Micb] of the program, or at runtime, by using the *LoadLibrary* function.

The task of loading a DLL in a process' address space is laborious and requires several steps, performed by two separate components: the kernel- and the user-space loaders. As their names suggest, the former is implemented in the operating system kernel, while the latter runs in the context of the process requiring the library. The DLL loading procedure starts with the kernel-space loader mapping the library in memory, following the layout specified in the *PE header*. The execution then reaches the user-space loader, which, in the case of position-independent code (the default for DLLs in Windows) applies the "relocations," i.e., patches the code of the library to ensure that the instructions accessing absolute addresses in memory behave correctly, even if the library is not mapped at its preferred base address. Since the Windows libraries have complex interdependencies, to load a library successfully the user-space loader also needs to find its dependencies and initiate the loading procedure for each

of them in a recursive fashion. Only when all the dependencies are correctly satisfied can the user-space loader run the initialization routine of the library, which creates and initializes the data structures that the library code needs.

*Ntdll*, the lowest-level library of the Windows operating system, provides the implementation of the user-spacer loader. This DLL also ships the user-space implemented *NativeAPIs* and the stubs to invoke those that the kernel offers through the system call mechanism.

To avoid a chicken and egg problem ("if the user-space loader loads DLLs and *ntdll* implements the user-space loader, then who loads *ntdll*?"), Windows maps this library in every process on the system. In fact, *ntdll* also provides the first routine executed in the context of a newly created process (*LdrInitializeProcess*), which is in charge, among other tasks, of creating the process environment block (*PEB*) and other critical data structures.

Alongside *ntdll*, modern versions of Windows also load *kernelbase* and *kernel32* in every process on the system. These two libraries provide those *WinAPIs* that a process likely needs. For example, they implement high-level functionalities for file manipulation, threading, and heap management.

### 4.2.3 API Sets and Umbrella Libraries

Windows supports other platforms than personal and server computers, such as embedded devices, game consoles, and virtual reality head-up displays. These devices often support only a subset of the *WinAPI* collection, which they implement in DLLs that follow a different naming scheme than those found on a regular PC.

In an effort to overcome *WinAPI* fragmentation, thus improving inter-device portability, recent versions of Windows introduced the concept of *API Sets* [Micc], i.e., groups of semantically similar *WinAPIs* that a device-specific version of Windows may or may not support. User programs can query the availability of an *API Set* on the system by loading the corresponding "umbrella library." If the loading process succeeds, the system effectively supports all the *WinAPIs* in the set, and the program can access them from the loaded library. Umbrella libraries follow a common naming scheme ("api-<feature>-l<major version>-<minor version>.dll") on all versions of Windows. Instead of containing the actual implementation of the *WinAPIs* in the corresponding set, umbrella libraries act as forwarders towards the system DLLs that do provide it.

Since Windows 7, the Windows system DLLs themselves started using *API Sets* to encode their dependencies. As a result, their import tables would often point to the umbrella library rather than the system library providing the API implementation. Unfortunately, this makes navigating the intricated interdependencies of the Windows library ecosystem even more challenging.

## 4.3 API-Based Behavioral Analysis: State of the Art and Bypasses

The ultimate goal of malware analysis is to distinguish malicious programs from legitimate ones by inspecting their behavior, that is, the ensemble of the activities they carry out when running on a system. In the Windows operating system, one way to encode a program's behavior is to list either the WinAPIs or the syscalls it employs.

From an analyst point of view, using the WinAPIs to encode the behavior of a program is preferable over using the syscalls: Their high-level semantic makes the WinAPIs more intelligible, allowing the analyst to distinguish malicious activities in less time and with minimal effort. Moreover, since they are more stable among major versions of the operating system, preferring the WinAPIs to their lower-level counterparts allows writing tools and analysis systems that are easier to port to new versions of the OS. Consider, for example, antivirus dynamic signatures: Writing them in terms of WinAPIs ensures that the same signatures will work on any version of Windows, protecting even new releases from known malware.

Broadly speaking, to list the WinAPIs that a program employs, one can either opt for a static or a dynamic approach. The first parses the binary's import table, which lists the functions it uses and the DLLs that implement them. The second, instead, consists of executing the program in a monitoring environment that logs every time the execution flow steps into a system library.

As is often the case for static binary analysis techniques, the first strategy falls short when dealing with obfuscation mechanisms. Obfuscated malware frequently hides or tampers the import table to make it harder to parse for static analysis tools. Some malware families go as far as reimplementing portions of the user-space loader to resolve and bind dependencies and do so in a custom and convoluted way. For this reason, recent research works have focused on recovering the import table from

obfuscated malware using dynamic analysis.

As we saw in Chapter 2, researchers also proposed a plethora of dynamic analysis techniques to trace the execution of a Windows program in terms of the WinAPIs it invokes.

Although worthy of interest in principle, all these techniques present technical flaws that make them less effective or even completely bypassable by the analyzed program. For example, *API hooking* can be bypassed by jumping ahead of the starting point of the function to invoke (after having crafted a coherent stack frame by "emulating" the skipped instruction), effectively avoiding any hook. Similarly, hardware-assisted tracing can be hindered by executing massive amounts of (otherwise meaningless) control flow-deviating instructions to fill the trace buffer with useless information.

Even implementation-specific bypasses are possible, like the Heaven's Gate technique [Cise] that exploits the way several antivirus products implemented *API tracing* for 32-bit programs running in compatibility mode on a 64 bit Windows OS. For this type of program, the antiviruses only monitored the 32-bit versions of the system libraries, completely overlooking their 64-bit equivalent, which the OS maps in the process address space and that the program can access after switching the processor mode of operation.

What makes the entire class of dynamic *API monitoring* techniques frail, however, is the ubiquitous assumption that the control flow must reach the system DLL. Only when this happens, in fact, do these techniques log the API invocation. While this assumption may seem reasonable and harmless, it opens the door to a more generic type of bypass strategy. An attacker can indeed avoid using the WinAPI altogether, relying only on the less semantic-rich NativeAPI to complete their malicious activities.

Unsurprisingly, researchers have reported malware families resorting to such a strategy. Notable examples are FormBook[Fir] and Flokibot [Mala]. Given that NativeAPI's ABI (e.g., the syscall number parameter) may change among different versions of Windows, samples of these families started implementing boilerplate code to invoke the kernel routines, which they would then fill with the syscall numbers extracted from the *ntdll* library stored on disk.

While this trend is alarming, as far as we are aware, malware has only been shown to implement this bypass strategy for the early stages of an infection, often only to inject malicious code in the context of a system process. Since the target process is considered trusted (thus, not moni-

tored), the injected code makes regular use of the WinAPIs to carry out the malicious activities.

We would argue that the reason behind the all-in-all limited adoption of this bypass strategy resides in the significant implementation complexity one incurs by only using syscalls. It is not surprising that process injection is, to the best of our knowledge, the sole documented use case for this strategy. In fact, implementing this technique does not require more than a handful of historically stable syscalls, like *NtVirtualAllocEx, NtVirtualWriteEx, NtCreateThread,* making it relatively portable, thus worth implementing from a malware author's point of view.

The next section discusses how this universal type of bypass can be pushed one step further, making it possible for a malware author to access high-level APIs (at development time) while still bypassing any API monitoring system (when the malware actually runs).

## 4.4 High-level API-Tracing-resistant Programming

The research question we tackle in this section is the following: How can a Windows program take advantage of the WinAPIs and bypass API monitoring solutions simultaneously? As Section 4.3 underlines, malware's attempts to circumvent API monitoring solutions have so far remained rudimentary (e.g., invoking syscalls directly). Such an approach does not scale: Re-implementing the same complex functionalities that the WinAPIs provide using only syscalls is out of reach because it requires extensive reverse engineering of the OS internals and great implementation efforts.

Our work tackles the problem from an orthogonal perspective. Instead of re-implementing a high-level layer of API, we adopt a *code reuse* approach that borrows the WinAPIs' implementation from the DLLs of the Windows operating system. The programs we are able to create with our technique are equipped with a *custom runtime* that contains all the WinAPIs the program needs. By doing this, these programs do not need to ever step into any system-provided modules, effectively bypassing any tracing mechanism that could be in place. One important note: differently than existing approaches, which borrow WinAPIs' implementation from the DLL files on the target host (e.g., a malware would read "kernel32.dll" from the victim's file system), our technique does not depend on these either. In other words, our technique allows a malware to *embed* the required resources (or, alternatively, make it possible to retrieve them at run-time from arbitrary locations, such as a network backend), instead of needing to trust resources on the victim machine — which could

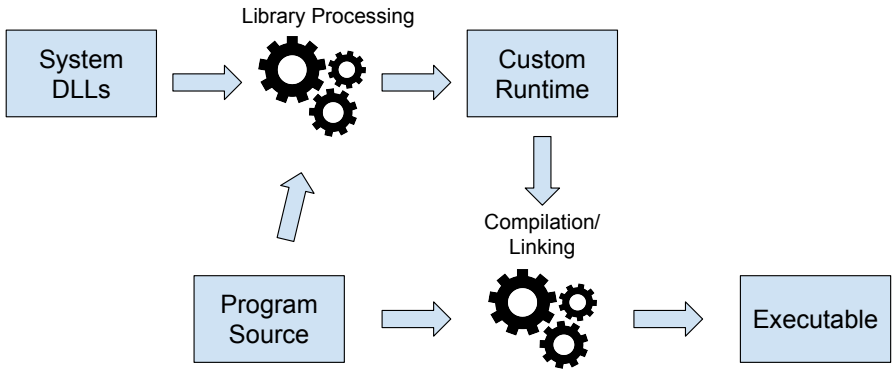


Figure 4.1: Overview of the offline phase of our approach

be a dynamic analysis sandbox. Although this difference seems minor, we needed to address a number of previously unsolved technical challenges. Moreover, this technique defeats analysis sandbox that "instrument" the actual DLL files on the analysis sandbox environments. This makes our technique more evasive than existing ones.

### 4.4.1 Overview

Our approach consists of two separate phases. The first one, which we call the "offline phase," takes place during the development of the program that is to be equipped with anti-tracing capabilities. Figure 4.1 shows the offline phase of our pipeline in broad lines. It starts with collecting the system libraries from a fresh installation of Windows. The collected libraries are then processed and merged into the *custom runtime*. Note how some knowledge about the program's source code is needed to build the runtime. This is due to the fact that, at least in the vast majority of cases, the program to equip with anti-tracing capabilities does not require the entire collection of libraries to function. The *custom runtime*, which is then linked to the final program, contains all the processed libraries and the logic to initialize them at runtime.

The second phase, which we call the "online phase," starts instead at the beginning of the execution of the equipped program. Before leaving the stage to the program's main function, in fact, the loader component of the custom runtime resolves the dependencies of the programs and fetches the corresponding library from the program executable. The custom runtime then loads and initializes each library in the context of the

current process, starting from *ntdll*, *kernelbase*, and *kernel32* that, as we will explain later in this section, need special care to behave correctly.

### 4.4.2 Technical Challenges

As simple as the idea may seem, this approach does not come without serious technical challenges.

In the first place, the Windows system libraries show complex interdependencies, which need to be considered when building the *custom runtime*. Failure to handle the DLL dependencies may result in unavailable functionalities or the execution flow inadvertently reaching the system libraries. The latter may happen, for example, when the custom runtime does not contain any of the dependencies of a library, and the loader resolves it to the one that the operating system provides. The widespread use of *API Sets* to encode dependencies among system DLLs introduces one more layer of complexity. These surrogate libraries do not contain the actual implementation, but act as placeholders that the system resolves at runtime. In other words, a library that imports an API-set library implicitly depends on a second one. To ship a stand-alone runtime, both implicit *and* explicit dependencies must be correctly handled.

Another major technical challenge of our approach lies in the three system libraries, namely *ntdll*, *kernelbase*, and *kernel32*, being already loaded and initialized when the startup phase of the *custom runtime* begins. Handling any of these three libraries the same way as the others results inevitably in them failing to initialize. Indeed, their initialization routines are not designed to run more than once per process. For example, for command-line programs, the *DllMain* of the *kernelbase* library, among other things, establishes a remote procedure call endpoint that has exclusive access to the console manager. If this routine were to run a second time for the same process, it would fail to establish a second endpoint.

Not running the initialization routines of these libraries at all is not an option either. Some vital data structures, such as the handle of the process heap, are created by these routines; Them being uninitialized correctly leads to inconsistencies and unexpected behaviors in many cases.

Lastly, we would argue that even if the initialization routines of these three libraries could run twice, initializing all the data structures adequately, this would still not be enough for the process to run correctly. In fact, having two distinct versions of *ntdll* mapped in the same process means having two different *PEB* structures, and since this data structure should be accessible through the `gs` segment register (which is how system

libraries access it), the *PEB* of the custom library must supersede the original one. However, modifying the memory area pointed by the `gs` segment register can lead to unexpected behavior challenging to troubleshoot. To overcome this problem, we opted for patching at runtime the three libraries' code, redirecting each memory access in their data sections, towards the data section of their system-provided counterparts. By doing so, we ensure that the libraries point to the correctly initialized data structures and that we do not create doppelganger *PEBs*.

The rest of the section provides more implementation details of both the "offline" and the "online phase," focusing on how we handled the challenges discussed so far.

### 4.4.3 Offline Phase

The offline phase of our approach pursues the objective of creating the *custom runtime*, starting from the system DLLs and linking it to the final executable.

This phase begins with finding all the system libraries on a fresh installation of Windows. To this end, we listed all the files with `dll` extension in the `System32` directory and its subdirectories recursively.

The next step of the process consists of finding the set of libraries on which the target program depends to assemble the *custom runtime*. This aims at reducing the size of the *custom runtime* by only selecting those libraries that the program would actually need during its execution. For this purpose, we gather knowledge about the WinAPIs that the target program needs and the DLLs that provide their implementation. For each of them, we compute the set of libraries they depend on by traversing their *import table* recursively.

*API Set* libraries need special care compared to standard libraries. In general, these libraries do not contain any code at all, but only an *export table* listing the WinAPI in the set. For each entry in the table, the *forwarder* field contains a pointer to the name of the library that effectively implements the matching function. We leverage this in our dependency processing to resolve *API Sets* to actual libraries.

Once we computed the set of dependencies of the program, we assign to each library a random name, ensuring to modify the import and export tables of each library so that they reflect the new naming scheme. Although this may resemble a naive form of obfuscation (and in many ways, it is), the primary purpose of this step is to avoid that loading any of the libraries in the *custom runtime* could result into loading system-provided

libraries. By default, on recent versions of Windows, the loader first checks whether the requested library is among the ones listed in the registry key `Session Manager KnownDLLs` [Mica]. Suppose that is the case, then the loader uses the system DLL instead of the one provided in the application's directory. Since several of the system libraries are listed in the registry key of a newly installed system, renaming them is a convenient way to force the system into loading the libraries of the *custom runtime* instead of the original ones.

During the offline phase, we statically analyze *ntdll*, *kernelbase*, and *kernel32* to list all the instructions that access their data sections. For the purposes of this work we only considered readable/writable data sections (e.g., ".data" and ".bss"), which are the ones containing all the important data structures. To achieve this, we used *IDA Pro* [HR] to disassemble and iterate over all the instructions in the libraries. Each time we encountered an instruction that used the *RIP-relative* memory addressing mode for one of their operands, we would add its address to the list of instructions to patch at runtime. At the end of the process, we encode this list in a table that we store in a dedicated section appended to the library itself. During the online phase, the custom runtime uses this information to patch the libraries.

The last stage of the offline phase consists of transforming each library into a resource that the final executable can access at runtime. Finally, we link the transformed libraries and the logic of the *custom runtime* that implements the "online phase" to the final executable.

### 4.4.4 Online Phase

The online phase of our approach starts at the beginning of the execution of the program equipped with the *custom runtime*. Its goal is to load and initialize the libraries that provide the WinAPI that the program uses. As explained before, we do not handle all the libraries in the *custom runtime* in the same way. While our *custom runtime* relies on the *LoadLibrary* API to load the majority of the libraries, for what concerns *ntdll*, *kernelbase*, and *kernel32*, we developed a technique that we call *Map&Patch*, designed to avoid loading twice the same libraries. The online phase starts by handling the three libraries before loading the rest of the *custom runtime*.

**Map&Patch.** Figure 4.2 depicts the inner mechanism of the custom loader that we implemented to handle the three libraries that the Windows OS maps before the process starts. For each of the three libraries, the custom loader retrieves the base address at which the system mapped their

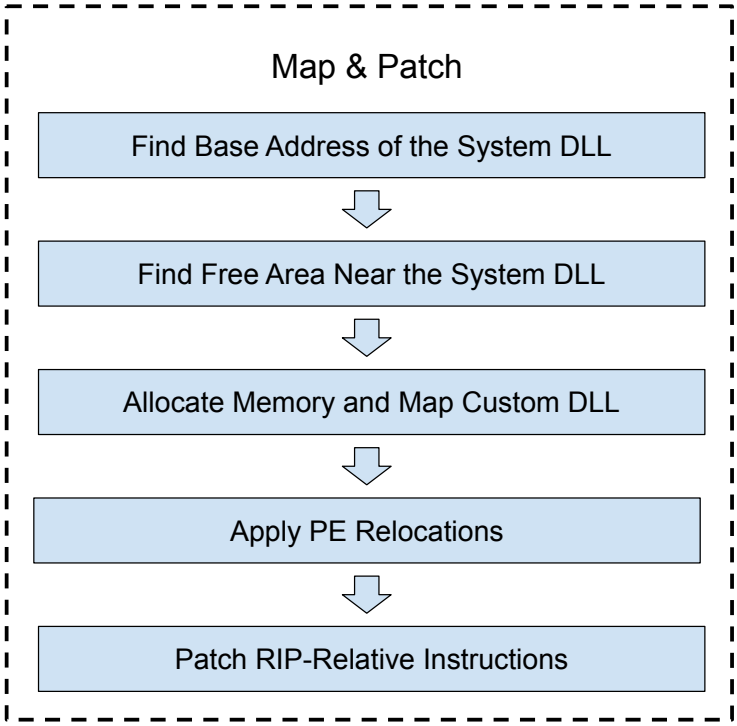


Figure 4.2: Map&Patch process for ntdll, kernelbase, and kernel32

original counterparts. To do so, we parse the `LoaderData` structure in the `PEB`, iterating over the `InMemoryModuleOrderList` array that contains information about every module loaded in the current process. Knowing the base address of the original library is paramount for two reasons. Firstly, to patch each instruction correctly, we need to add the offset between the system DLL's address and the custom one's to its *RIP-relative* operand. Secondly, the original library's base address limits the choice for the base address of the custom library. Indeed, if the two modules were too far away in memory, it would not be possible to patch the custom one appropriately because the Intel x86-64 ISA only allows addressing memory in a limited range around the address of the instruction ($\pm 2$ GB). For this reason, the next step of the *Map&Patch* technique is finding a suitable address for mapping the custom library.

We achieve this by using the `NtVirtualQuery` syscall, which, given an address, returns information about the memory region around the address, specifying its size and whether it is free or reserved. Starting from the address right above the end of the system library, we iterate the process until we find a free memory region big enough for the custom library. If the iterative process goes beyond an offset of `0x80000000` (the limit supported by the ISA for instruction pointer-relative addressing), we start it again from the address right below the base address of the original DLL and descending. If even this attempt of finding a suitable address fails, the process stops. We note, however, that this is very unlikely to happen in practice. In fact, at the beginning of the execution, only a tiny amount of memory is mapped. Even if everything coalesced in the same region, it would not occupy a 2GB range around the original libraries. In our experience, we never even needed to query the memory below the original library to find a appropriate address for the custom one. After finding one, the custom loader reserves the memory using the *NtAllocateVirtualMemory* syscall before mapping it according to the memory layout outlined by the *section table* in its PE header.

The *Map&Patch* process then applies the relocation information that the PE header provides. Supporting relocations is essential since the read-only sections of the three libraries contain pointers to data structures that need to be updated at loading time. In our experience, failure to implement relocations results in memory access violations once the program invokes any WinAPI using the custom runtime. The custom loader then patches the instructions found during the offline phase, adding the calculated offset to the *RIP-relative* operand of the instruction. To this end, it parses the table appended at the end of the library, in which each entry

points to a 4-byte word in the code section, corresponding to the operand to modify. Finally, the *Map&Patch* process ends by fixing the memory permissions of the custom library, as specified in its section table, allowing the program to execute the instructions in the code section.

**Loading other libraries.** After the *Map&Patch* process finishes, the program can already access various WinAPIs, including the primitives for file management and library loading. The *custom runtime* employs these features to create one file per library, according to the naming scheme decided during the offline phase.
It then proceeds to load each library invoking the *LoadLibrary* WinAPI that the custom version of *kernelbase* provides.

At this point, the *custom runtime* is fully initialized, and the program can start.

### 4.4.5 Proofs of Concept

To demonstrate the feasibility of our approach, we designed three proofs of concept that carry out tasks typically found in malware. Each PoC was implemented in less than 80 lines of C++ code, only a few more than it would be required to implement the same functionalities in a regular Windows program.

The first (and simplest) PoC use file manipulation APIs to open, write, and read data from a file on the disk.

The second carries out a process injection employing the *QueueUserAPC* WinAPI, using a technique similar to the one described by [Red] and commonly used by malware, including *APT* [Cybc].

The last PoC performs network operations using WinAPIs provided by the *wininet* DLL. This PoC is the most complex among those we developed, and it carries out an *HTTPS* request, retrieving a web page from an URL.

## 4.5 Towards Reconstructing API-Level Information from Syscalls

The bypass approach we presented in the previous section and our the state-of-the-art investigation highlight how fragile and unreliable WinAPI-based behavior analysis truly is at its core. The source of all evils, in this case, is easy to spot: WinAPI-based analysis techniques try to fight

the enemy on its own terrain. These techniques rely on information collected from within the malicious process' context and the address space, which are, however, under the complete control of the malicious process — there is no guarantee that the information collected from it is genuine.

A more sound approach to malware behavior analysis should rely on information collected at the boundaries of the context of the malicious application, where the malware can exercise only limited control. In most scenarios, the syscall layer between the user space and the kernel is the most natural of the interfaces to monitor because, under a reasonable attacker model (i.e., uncompromised kernel) we can consider this layer trusted.

However, the fact that collecting WinAPI-level information is unreliable does not make it less valueable. Indeed, the higher-level semantics that these APIs carry make them easy to interpret and employ for describing complex behaviors. The same cannot be said for syscalls.

Driven by the rationale that, under the hood, the WinAPIs make use of the syscall layer, we set to explore the feasibility of reconstructing their high-level semantics from a syscall trace. To this end, we dynamically analyzed a dataset of (non-malicious) Windows executables, recording both the WinAPIs and the syscalls they invoked, so to build a knowledge base. Our measurements aim at quantifying the complexity of the "mapping problem," i.e., the process of bridging the semantic gap between WinAPI and syscalls.

### 4.5.1 Analysis Environment

To collect WinAPI and syscall traces, we developed a dynamic analysis tool based on Intel Pin [Int], the DBI framework. This framework grants the ability to instrument a program at several granularities, such as function calls, basic blocks, or even single instruction levels.

Intel Pin also allows intervening anytime certain events happen during a program's execution. For example, it is possible to invoke user-provided callbacks each time the process loads a module. We used this feature to intercept the loading of any system library and to instrument the functions they export. Doing so guarantees that anytime the program invokes a WinAPI, Intel Pin executes our custom routines both before the execution of the API and after it returns. In this way, we can log both the beginning and the end of each WinAPI.

Our tool employs a similar process to trace the syscalls too. *ntdll*, in fact, provides wrappers for all the syscalls that the kernel supports and

that the other system DLLs use for their purposes. By instrumenting this library at the beginning of the execution, we are able to run our custom logging infrastructure anytime the WinAPIs reach the syscall layer.

Noteworthy, our tool enforces strict temporal ordering of the entries in the traces and logs both the WinAPIs and the syscalls on the same channel. These two characteristics make it possible to infer a caller-callee relationship between the entries in the trace, allowing to obtain the syscalls that the program invoked as a side-effect of employing a WinAPI.

We compiled our tool for the *Intel x86* and *x86-64* architectures to support both 32 and 64-bit executables. The two versions were deployed on a group of virtual machines — each provided with 2GB of RAM and two logic CPUs — running on a server powered by four Intel Xeon Platinum 8160. Each VM runs a vanilla installation of Windows 10 (version 1909), which we restore to a clean snapshot at the end of the execution of each sample in the dataset. We instructed our tool to run each sample for 5 minutes.

### 4.5.2 Dataset

Our dataset contains a total of $2.3117 \times 10^4$ non-malicious samples. We relied on Chocolatey [cho] for the creation of this dataset. This third-party package manager allows installing a multitude of free software products quickly. We installed all the available packages on Chocolatey on a machine running a clean version of Windows. We then selected all the files on the system with *.exe* extension. As a result, our dataset also contains executables that Windows itself ships.

The strict policies that Chocolatey enforces before supporting new packages guarantee that the entirety of this dataset is non-malicious — a desirable property for the large-scale analysis we conduct. Since our objective is to characterize the relationship between WinAPI and syscall invocations, we are only interested in recording the syscalls that the program invokes indirectly by employing the WinAPIs. Malware, on the other hand, often uses syscalls directly, which could pollute our results, making it deleterious for our purposes.

### 4.5.3 Data Processing

The traces that our DBI tool captures contain one record for each monitored event, i.e., the invocation of either a WinAPI or a syscall wrapper. For each event, the system logs the API/syscall name, the library providing it, and whether the event corresponds to the beginning or the end of

```
1 NtOpenThreadToken
2 NtOpenThreadToken
3 NtOpenThreadToken
4 NtSetInformationThread
5 NtAlpcCreateSecurityContext
6 NtSetInformationThread
```

Listing 4.2: Syscall Trace of wlanapi.dll:WlanOpenHandle

the execution of the target function. Our tool also logs the stack pointer and the first element in the stack, i.e., the return address of the current invocation. This data is helpful in supporting recursive functions during the trace processing phase.

We processed the raw data collected during the dynamic analysis campaign and organized it in an easy-to-query data structure (from now on the *knowledge base*) that stores the list of syscalls generated by each WinAPI that the samples invoked. For this work, we are only interested in the WinAPIs that the programs invoked directly. In other words, even if our tool recorded intermediary APIs too, the data processing phase kept only the top-level ones. The rationale behind this choice is that it allows us to keep only those WinAPIs that the programs intended to use.

The data processing treats the trace recorded for each sample sequentially. For each of them, the process commences by separating the entries based on the threads that invoked the corresponding WinAPI or syscall. Then, for each thread, the analysis system slices the corresponding trace in chunks delimited by the start and end event of the same API.

For determining that a start event and an end event belong to the same WinAPI invocation, the system checks that the API and library names are the same and that both the stack pointer and the return address match. These comparisons prevent matching the beginning of a recursive API with the end of any reentrant calls. For each chunk, the system stores the invoked syscalls in the knowledge base as a new entry for the WinAPI.

### 4.5.4 Examples of API-Syscall Mapping

By querying our knowledge base, it is possible to find different types of WinAPI, performing all sorts of tasks, from file manipulation to graphic user interface creation, wireless access point discovery, and internet-related operations.

```
1 NtDuplicateObject
2 NtCreateEvent
3 NtDeviceIoControlFile
4 NtDeviceIoControlFile
5 NtDeviceIoControlFile
```

Listing 4.3: Syscall Trace of ws2_32.dll:setsockopt

The complexity (in terms of generated syscalls) of the WinAPIs in our knowledge base varies greatly. While some WinAPIs generate syscall sequences that are expected and intelligible even to the untrained eye, others are exotic and counterintuitive.

Let us introduce a few telling examples of the wide range of complexity among the WinAPI we recorded.

The *CreateFileA* and *WriteFileA*, both implemented in *kernelbase*, respectively invoke one syscall each, namely *NtCreateFile* and *NtWriteFile*. In other words, these APIs map directly to one syscall, which is also the semantically closest to the task that the corresponding API performs. From a semantic reconstruction point of view, APIs such as these two are trivial (and almost useless) to recover.

Other WinAPIs' semantics are remarkably more difficult to reconstruct. Take *WlanOpenHandle* and *setsockopt*, whose syscall footprints are shown in Listings 4.2 and 4.3, respectively. The former instantiates the wireless LAN manager, while the latter is the standard C function that sets options on a network socket. In the perspective of semantic reconstruction, the differences between these two APIs and the previous examples, do not lie exclusively in the length of the syscall traces they produced. The syscall footprints of *WlanOpenHandle* and *setsockopt*, in fact, do not convey any information that intuitively hints to the original APIs' task.

What makes these two WinAPIs' syscall trace so unintelligible is their use of the *advanced local procedure call* [RSI12a, p. 209] (ALPC, from now on) in the case of *WlanOpenHandle*, and device driver input-output control functions (*IOCTL*) [RSI12b, p. 25] for *setsockopt*. ALPC employs a few dedicated syscalls (one of which is `NtAlpcCreateSecurityContext` in Listing 4.2) to implement a request-response protocol, through which user-space processes can communicate with the system services. On the other hand, device drivers execute IOCTL functions when the user-space process invokes `NtDeviceIoControlFile` (as in Listing 4.3) or other specific syscalls.

These two mechanisms reflect the modular design of Windows: For most tasks, the kernel only acts as a message delivery facility rather than carrying out the actual work, in a microkernel architecture fashion[TB15, p. 65]. Consequently, the Windows syscall layer is rather small and message-passing-oriented, compared to other operating systems that expose, instead, a thicker and more task-oriented one[2].

### 4.5.5 Preliminary Measurements

In total, we recorded $1.1552 \times 10^4$ distinct WinAPIs and $184$ distinct syscalls. Our dynamic analysis campaign captured a total of $2.018\,345\,73 \times 10^8$ WinAPI invocations (the average number per WinAPI is $1.7472 \times 10^4$) and $1.007\,887\,7 \times 10^7$ syscall invocations ($5.4776 \times 10^4$ invocations per syscall, on average).

By manually exploring the knowledge base, we noticed that certain syscalls are prevalent in the traces recorded for many WinAPIs, more specifically, `NtWaitForSingleObject` and `NtAllocateVirtualMemory`. Most of the times, these syscalls did not seem fundamental to the overall function and semantic of the WinAPIs that employ them. The first one, for example, is used for thread synchronization, which, by itself, does not carry any information about the API behavior. The second allocates new memory in the process address space. In the vast majority of cases we inspected, its invocation seems to be a byproduct of the process' state at the moment of the WinAPI invocation. This syscall is indeed also used to reserve new memory pages for the process' heap when the already allocated ones are full. This is to say that any WinAPI that creates heap objects can invoke this syscall if the process is running out of heap space. A third example of widespread syscall is `NtClose`, which disposes of any type of object created by the Windows operating system, including open files, local procedure calls endpoints, and internet connections. This lack of specialization is what makes, in our opinion, this syscall marginal from a semantics point of view. Since, in most cases, these syscalls do not carry any valuable information about the WinAPI that invoked them, we decided to consider them as noise and to remove all their invocations from our knowledge base. From now on, all the data we report are computed without considering these three syscalls.

One fundamental limitation prevents retrieving the entire WinAPI trace of a program's execution from the corresponding syscall trace: Some APIs do not have any syscall footprint. In fact, whole classes of

[2]For reference, the Linux operating system has a dedicated syscall for `setsockopt`

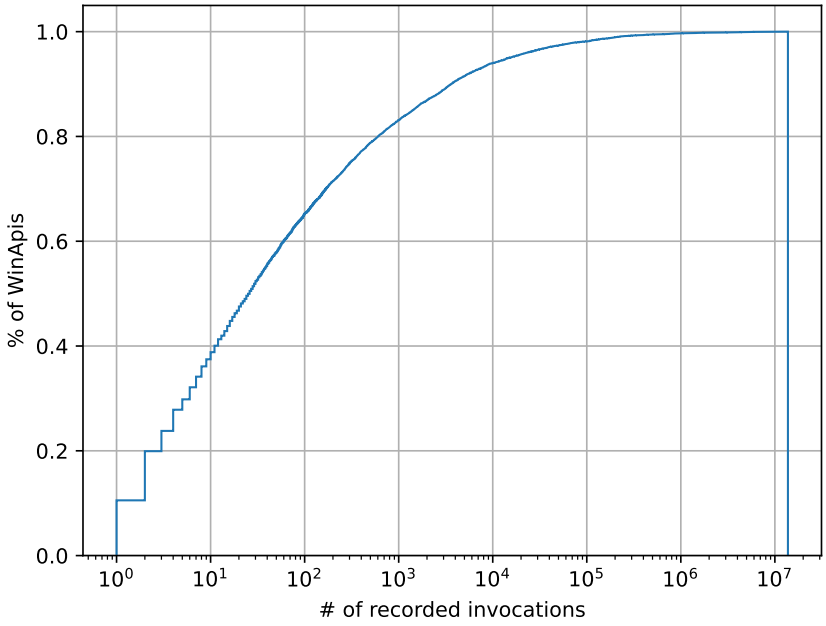


Figure 4.3: Cumulative distribution of the WinAPIs over the number of recorded invocations

WinAPIs (e.g., string manipulation) are entirely implemented in user-space, thus do not invoke any syscall. In our knowledge base, we counted $9.936 \times 10^3$ WinAPIs ($86\%$ of the total) that did not result in any syscall across any of their recorded invocations.

As one would expect, programs employ some types of WinAPI more than others. Consequently, during our experiment we recorded many invocations of the same WinAPIs and very few of some others. Figure 4.3 presents the cumulative distribution function (CDF) of the WinAPIs over the number of invocations we recorded during the experiment. As the graph shows, we recorded less than ten invocations for circa 40% of the WinAPIs, while the top 10% of the APIs have been recorded more than three thousand times. If we do not consider those WinAPIs that never invoke syscalls (see Figure 4.4), the situation is slightly different. On average, those APIs that invoke syscalls were recorded more times: the share of WinAPIs that we observed ten times or less is circa 27%, while the most recorded 10% was observed at least 13 thousand times.

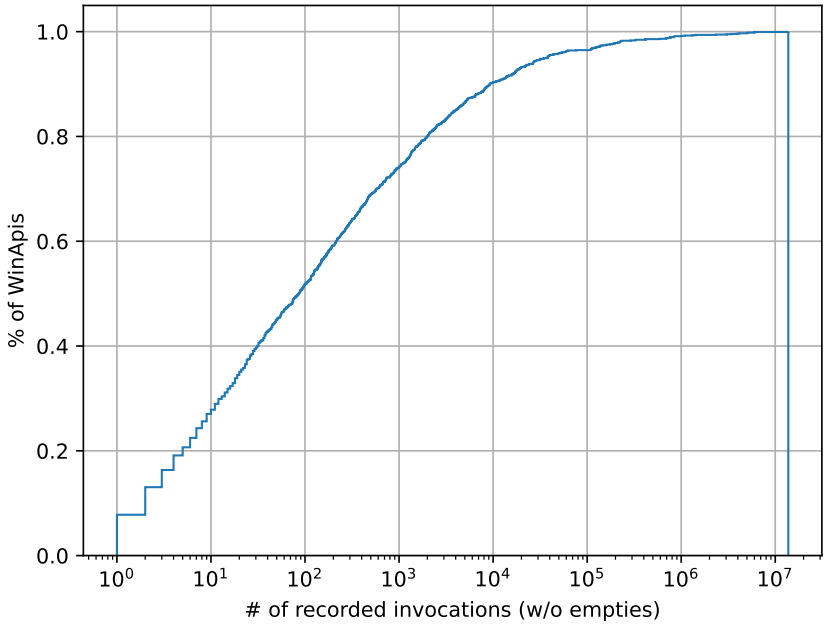


Figure 4.4: Cumulative distribution of the WinAPIs (excluding those that do not invoke syscalls) over the number of recorded invocations

### 4.5.6 Intra-API Similarity

Measuring the similarity between the syscall trace resulting from the invocation of a WinAPI can suggest the actual complexity of bridging the semantic gap between the two layers. Suppose having an unknown sequence of syscalls from which to reconstruct the candidate APIs likely to have generated them. If invoking the same WinAPI always resulted in the same syscall sequence, this task would be relatively easy because we would need to compare the syscall sequence to the sole "footprint" of each API.

As we frequenlty see in our knowledge base, however, two invocations of the same WinAPI often result into diverging sequences of syscalls. One interesting measure of the complexity of each WinAPI is the number of different syscall sequences that it can generate. Figure 4.5 shows the CDF of the WinAPIs with respect to the number of distinct syscall sequences that we recorded during their invocations. As the graph reports, the median number of syscall sequences per API is two, and about 88% of the APIs

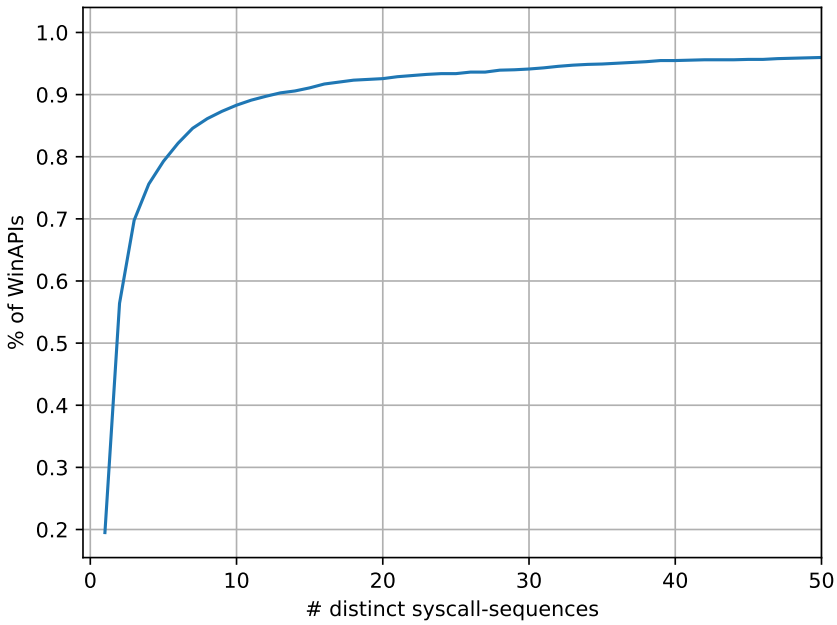


Figure 4.5: Cumulative distribution of the WinAPIs over the average length of the syscall traces they produced

produced ten distinct sequences or less.

The bare number of distinct syscall sequences, however, tells only part of the story. Indeed, this measure does not consider that APIs may generate specific syscall sequences very rarely or under abnormal conditions. For example, an API that always generates the same syscall sequence under normal conditions could fail and return an error before invoking any syscall at all if it received the wrong parameters. To take cases like these into consideration, we measured the *normalized entropy* of the set of syscall sequences of each WinAPI as follows:

$$\eta = \sum_{i=0}^{n} \frac{p(s_i)\log_2(p(s_i))}{\log_2(n)}$$

where $p(s_i)$ denotes the "probability" of i-th distinct sequence recorded for the WinAPI in question, that is, the ratio between the number of times we recorded that sequence and the total number of invocations of that API. The normalized entropy provides qualitative information about the diver-

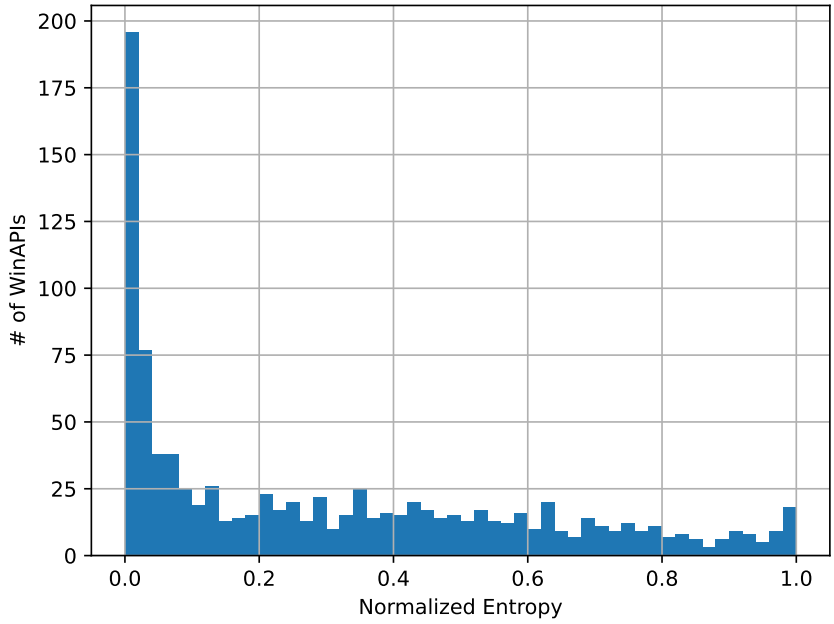


Figure 4.6: Disribution of WinAPI over the normalized entropy of their traces

sity of the sequences an API can generate. A lower value of $\eta$ indicates a certain "stability" in the sequences that the API generates. In contrast, a high value suggests that the API behaves in a in a way that is trickier to predict. Taking it to the extremes, a value of $\eta$ of 0 means that the WinAPI always generate the same syscall sequence, while a value of 1 (the maximum possible value of normalized entropy) means that the API can produce more than one sequence, each of them with the same probability. We calculated the normalized entropy for each WinAPI, for which we recorded at least 30 invocations and that produced syscalls at least once. Figure 4.6 shows the distribution of the normalized entropy we calculated. The left-side peak represents those APIs whose normalized entropy is close to 0 and thus always produce the same syscall sequence. In general, the normalized entropy of the dataset seem relatively low, suggesting that, on average, WinAPIs are relatively stable in terms of the syscall sequences they produce.

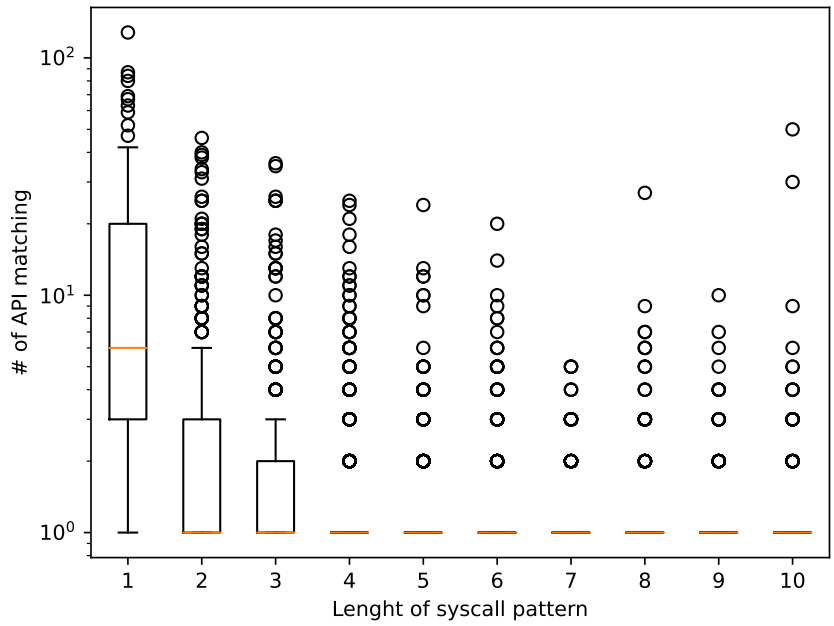


Figure 4.7: Number of matching APIs per syscall pattern (of length from 1 to 10)

### 4.5.7 Inter-API Simiilarity

The likeness among syscall traces generated by the same API is not the only indicator of the reconstruction problem's complexity. An orthogonal metrics consists in estimating the degree of similarities between the traces produced by different APIs. As an experiment of thought, take two APIs that exclusively and always generate the same syscall trace. Despite being both very stable, for all intent and purposes, they are completely indistinguishable from the point of view of the reconstruction process.

To measure the degree of similarities among the WinAPIs, we gathered all the sequences shorter than or equal to 10 syscalls in our knowledge base. For each sequence, we then counted the number of WinAPIs which had generated it at least once. This measure quantifies the difficulty of distinguishing which API generated an unknown syscall sequence: A high number of WinAPIs matching the same sequence indicates that the reconstruction problem is intrinsically ambiguous.

Figure 4.7 presents the results of the measurements. For each se-

| Length | Tot. Sequences | Overlapping Sequences (%)) |
|---|---|---|
| **2 syscalls** | 467 | — |
| **3 syscalls** | 675 | 631 (93.5%) |
| **4 syscalls** | 745 | 726 (97.4%) |
| **5 syscalls** | 681 | 674 (99.0%) |
| **6 syscalls** | 636 | 627 (98.6%) |
| **7 syscalls** | 577 | 580 (98.8%) |
| **8 syscalls** | 522 | 521 (99.8%) |
| **9 syscalls** | 495 | 495 (100%) |
| **10 syscalls** | 433 | 431 (99.5%) |

Table 4.1: Overlapping sequences per each length class

quence length, the figure shows one boxplot representing the number of WinAPIs matching for each syscall sequence of that length. The orange stripe in the middle of the box indicates the median value of the number of APIs matching, meaning that 50% of the sequences of that length match with that many APIs or more. The lower and upper borders of the box indicate the first (Q1) and third quartile (Q3), respectively, while the whiskers mark a one and a half interquartile range above and below Q3 and Q1, respectively. The circles above the whiskers represent outliers matched by many more APIs than the average.
The graphs suggest that the longer the syscall sequence, the less ambiguous it is, as fewer WinAPIs can produce them. Intuitively, this seems reasonable: longer syscall sequences are more specialized, which in turn means that only a few highly specialized WinAPIs can invoke them.

None of the measurements presented so far highlighted an unsurmountable obstacle that hinders the reconstruction of WinAPI-level semantics from syscalls. As a matter of fact, not only the syscall sequences produced by the same API are relatively similar, but also, notwithstanding few exceptions, each syscall sequence matches only a few APIs or, in most cases, a single API.

So far, however, we have considered only single syscall sequences, and attempted to find which API may have invoked them. This simplified scenario does not model what handling an entire syscall trace is. By processing one sequence at a time, we implicitly assume that, on a syscall trace, we can identify the start and endpoint of each API. Unfortunately, this is never the case in a real-world scenario.

What we have yet to consider is that the syscall sequences in our knowledge base can partially overlap. Incidentally, we believe this is what makes

the reconstruction process extremely challenging in real scenarios. Table 4.1 reports the number of overlapping syscall sequences in our knowledge base, divided by length. We only considered sequences whose size is greater than two syscalls, but less than ten. We labeled a sequence as "overlapping" if it contains any known shorter sequence. The results of these measurements unveil how ambiguous the syscall sequences really are. For each length class, overlapping sequences represent 90% of the total. Notice that we purposely did not consider the 1-syscall long sequences in the knowledge base. If we were to consider those in the measurement, the number of overlapping sequences would be even more significant.

## 4.6 Discussion

This section discusses the applicability in real-world scenarios, and the implications of our work, defining a roadmap for the security community to address the flaws we discussed in this paper. We start by describing the drawbacks that might limit the adoption of our bypass approach by real-world malware. We then move to examining the classes of security-oriented software that need to be adjusted and how. Finally, we survey possible avenues to further investigate the API semantic reconstruction from syscall traces.

### 4.6.1 Applicability of Our Approach in Real-World Programs

Given that we only implemented our approach in proof-of-concept form, one legitimate question is whether malware could use this mechanism for evasive purposes.

A first concern could regard the out-of-the-norm size of the executables that our procedure generates. Even to implement simple functionalities, programs compiled with our approach easily reach tens of megabytes in size, which is orders of magnitude bigger than regular malware [UPGB19b]. However, this argument does not stand because regular software often reach several tens of megabytes in size, meaning that a big size would not be a distinctive characteristic of the executables compiled with out approach. Indeed, malware analysts have reported that some malware families, for example LoudMiner, embedded entire virtualization stacks in their samples, suggesting that they care more about stealthiness than size [ESE]. Furthermore, by applying more binary analysis to the Windows DLLs, it could be possible to reduce even more the size of the executables by pruning all the code in the custom runtime that is not

vital to the functionalities the program uses. Another alternative is that these payloads could be obfuscated and embedded as "PE resources", or even downloaded from network end-points at runtime.

Another possible concern about the current implementation of our approach is that it generates version-specific executables. This is due to the custom runtime being based on version-specific Windows libraries. While we acknowledge this as a valid argument, we do not believe that producing version-specific programs is a fundamental flaw of our proposal. In the first place, our approach in its current version could be used for malware running only specific versions of the Windows operating system, for example for targeted attacks. Moreover, with minor modifications, our approach can be used to embed more versions of the custom runtime in the same binary. During the *online phase*, then, the program would choose which runtime to employ based on the version of Windows on which it runs.

We also envision a common solution for both problems described above. With enough development effort, it is possible to setup a sort of "custom-runtime delivery networks," through which malicious actors can provide their malware with the custom runtime. To support this mode of operation, one could implement a preliminary stage in the *online phase* to discover the current version of the operating system, and then download the corresponding custom runtime from the network. Since the executables developed with this approach do not need to embed any custom runtime, their size will be comparable to that of the average malware sample.

A third critic could point out that a program containing known Windows libraries is already suspicious. To put it another way, one could apply the simple heuristic of marking executables embedding any Windows library as malicious. This argument has its roots in the observation that malware is indeed the only class of software with enough incentives to adopt such an approach, i.e., regular non-malicious software will never embed the Windows libraries in its executables. While this heuristic can work on the current version of our approach, nothing prevents malicious actors from obfuscating the produced executables by packing or crypting, effectively defeating any attempt to recognize the Windows libraries statically.

### 4.6.2 Beyond API-level Encoding of Malware Behavior

As documented in Section 4.3, current API-monitoring solutions have fundamental flaws that allow attackers to bypass the instrumentation, effec-

tively hiding their malicious actions. While it was known to the security community that these API-based solutions are bypassable, this paper shows that automating and "scaling up" the evasions is more feasible than previously thought.

The bypass strategies that malware has adopted so far are naive and cumbersome, limiting their applicability to simple tasks. However, we showed that a more "programmer-friendly" approach to API-monitoring bypass is not only possible, but also completely within reach of malicious actors. We believe that it is only a matter of time before analysts start encountering malware shipping more sophisticated and universal bypass techniques.

To mitigate this upcoming threat, the security industry needs to renew its current approach to malware behavioral analysis, moving from an API-centric conception towards a more resilient and harder to bypass syscall-centric one.

End-point protections such as EDR and antiviruses raise particular concern. Too often these tools employ user-space techniques for API tracing, in order to help preventing malware infection at runtime. These approaches, which the red teaming community already considers frail [Mos, Bui], will be less effective in facing this type of threat.

Finally, dynamic analysis sandboxes would need to be revised too, since several products of this class have behavior analysis based on API-tracing as one of their main selling points. In fact, even the most sophisticated techniques (e.g., emulation, virtual machine introspection) cannot guarantee sound API traces.

### 4.6.3 Considerations on the Semantic Reconstruction Problem

The measurements exposed in Section 4.5 highlight that the task of recovering the *WinAPI-level* information from a syscall trace is complex and nuanced. Despite remaining an open problem, reconstructing high-level semantic from low-level (but trusted) sources still stands as our best chance to counter malware that has more and more incentives to conceal its behavior from analysis systems and antiviruses.

The difficulty that lies at the heart of this problem, is the inherent ambiguity that syscalls carry. As our experiments showed, different APIs use the same syscalls, often in the same order, making them tricky to distinguish in practice.

We believe that the key to solve the semantic reconstruction problem in the general case is to reduce the sources of ambiguity.

One possible strategy could be to aggressively prune noisy syscalls, in a similar way to what we did in our experiment. Removing those spurious syscalls whose invocation is a mere byproduct of low-level mechanism (e.g., synchronization, heap management) can reduce the traces' ambiguity. One could go as far as ignoring any syscall that is not security sensitive. This, however, comes at the price of giving up reconstructing those APIs that rely solely on the ignored syscalls.

Another avenue to simplify the reconstruction problem can be to inspect, at analysis time, the parameters of the syscalls — especially those involved in local procedure calls. As we described in Section 4.5.4, *WinAPIs* use ALPC when they need to access a service provided by a system user-space process. This is the case of several *WinAPIs*, including network operations and cryptographic primitives. Unmarshaling the data exchanged between the analyzed process and the service provider yields less ambiguous information, closer in semantics to that of the corresponding API. CopperDroid [TKFC15b, RFC13] adopts a similar approach to recover semantic data from binder transactions, roughly the equivalent of ALPC for the Android operating system.

## 4.7 Related Work

Over the last decades, as the malware threat grew, the security community has responded by thoroughly studying trends and refining analysis techniques for delivering the best possible countermeasures to the end-user. In this section, we analyze the closest previous researches to ours, categorizing them into four groups.

**Employing API traces for behavior analysis.** By virtue of the high-level semantic they carry, *WinAPIs* found a prominent role in encoding malware behavior. For example, researchers employed API-level information to characterize the behavior of spyware [KKB+06], botnets [SM07], and, more recently, ransomware [HBZ18]. Ki et al. [KKK15] proposed employing *WinAPI* sequences as an indication of compromise. Recent works by Rabadi et Teo [RT20] and Pirscoveanu et al. [PHL+15] even used API-level information for machine learning-based malware detection. Despite their unquestionable contribution to the field, all these work rely on correct API-level information, which unfortunately cannot be always trusted. As we described in this paper, no mechanism ensures capturing API traces in a reliable way. To make things even worse, malware authors are already moving towards not using APIs at all.

**API-level information collection.** To the extent of our knowledge, the first attempt at API tracing dates back to 1999, when Hunt et Brubacher [HB99] introduced the technique, today known as API hooking, that many security solutions employ, such as dynamic analysis sandboxes [OM13] and EDRs [Com]. The research community has also proposed several other mechanisms to collect API-level information in different contexts. For example, various works [KISH13, CML+21, CMF+18] refined import table reconstruction from obfuscated malware. As we covered in more detail in Section 4.3, all these previous attempts to gather API information implicitly assume that the analyzed malware employs the *WinAPIs*, which it is not necessarily the case.

**Documenting bypassing techniques.** Throughout the years, the security industry made a considerable effort to report new trends in malware evasion techniques. Several blog posts by both antivirus companies and malware analysts detailed malware families evading API hooking by employing direct system calls technique [Mala, Fir, Cybd, Cyba, Malb], and sketched mitigations against it [Cybb].

ScareCrow [Sec] provides anti-hooking functionalities by modifying the code of the system libraries loaded in memory at runtime. In particular, ScareCrow reads the preamble of the functions it intends to use from the system DLL on the disk and re-writes them in memory, overwriting the hooks that EDRs may have introduced. This approach assumes that the DLL on the disk is genuine, which is not necessarily true. Analysis tools may, in fact, modify the library on the disk.

Kawakoya et al. [KSO+17] proposed StealthLoader, a custom user-space loader that programs can use to map system libraries, theoretically defeating API monitoring solutions. Despite resembling ours, this approach presents several limitations that we overcame. In the first place, StealthLoader also makes the wrong assumption that the libraries on the disk are genuine. Our approach does not have this limitation: By embedding its own custom runtime, a program compiled with our technique does not need to trust the DLLs found on the system. Furthermore, StealthLoader does not include any mechanism to cope with libraries already loaded at the beginning of its execution. As the authors acknowledge, this limits the number of *WinAPIs* that StealthLoader supports. For example, StealthLoader cannot support APIs that manipulate heap objects, which most programs use.

**Behavior reconstruction from syscalls.** Previous works acknowledged the inherent ambiguity in syscall traces and attempted to recover malicious behavior from them. Accessminer [LBK+10, FLBK15] modeled what

benign syscall interactions looked like and employed them for malware analysis, using an anomaly-based detection strategy. In the context of the Android ecosystem, previous works have attempted to reconstruct the API semantics from syscall traces. Copperdroid [TKFC15b, RFC13] recovers Android Service invocations, by unmarshaling the data that the application under analysis and the service managers exchange through the *ioctl* syscall.

# Chapter 5

# Towards Reconstructing API Information from Syscalls

***Exploring the Semantic Gap between APIs and Syscalls in the Android Operating System***

## 5.1 Introduction

In the realm of malware analysis for Android apps, dynamic analysis approaches and instrumentation techniques are at the foundation of virtually all existing analysis frameworks, developed by both academia and industry [GZZ+12, ZWZJ12, FADA14, ARF+14, GKP+15, LNW+14, WL16, HTP15, Dev17, fri]. While dynamic analysis approaches can take many forms, they all share one key aspect: given an application, the goal is to "capture" all actions it performs during its execution. To this end, these apps are run in an instrumented environment, which records a trace of the app's behavior.

**API-level tracing.** In Android, most of these approaches aim at producing a list of high-level API calls performed by the app under analysis. These high-level API calls are Java methods exposed by the Android framework, a vast extension of the Java SDK. These methods include standard Java methods (e.g., string operations, networking primitives), as well as a large corpus of Android-specific methods, such as APIs dealing with building Android user interfaces, inter-app interactions, reading values from device's sensors, sending and receiving text messages. Having access to an accurate trace of invoked APIs is of great importance. In fact, these APIs

capture the high-level, *semantic-rich* behavior of an app and allow both human analysts and automated approaches to detect and characterize both malicious and unsafe actions the app could perform.

These approaches work by heavily instrumenting the app itself or the execution environment (e.g., by function hooking). Unfortunately, to date, *all* current API-level instrumentations can be easily detected and bypassed [ABF+16]. The key problem is that these instrumentation mechanisms all introduce visible instrumentation artifacts, which coexist within the same security boundary as the app itself. Acquiring these high-level traces require heavy instrumentation, which is hard to implement efficiently and it is trivially detectable by malware, which could decide not to show any malice when instrumentation is detected [YIT+16]. Android apps can also contain components written in native code, whose behavior cannot be captured if the app's instrumentation is only performed at the Java API level. More importantly, previous works have shown that the mere presence of native code can make the results of the analysis of the Java layer not only detectable and evadable, but even misleading [ABF+16]. In fact, since native code components and Java code run within the same security boundary, native components could surreptitiously modify the intended functionality of Java components making high-level recordings of an app's behavior completely unreliable. These issues severely affect the reliability of acquiring high-level API-based traces.

**Syscall-level tracing.** A different approach consists in capturing the actions performed by an app by recording its low-level interactions with the operating system, specifically, by recording the system calls (syscalls) it invokes [TKFC15a, DAUR16]. This approach is not affected by the limitations mentioned above. In fact, regardless of the language used to implement the different app's components, to interact with the operating system, the app *needs* to eventually invoke a system call. Moreover, this kind of instrumentation is harder to detect, since it can be easily implemented entirely by code running in kernel mode, not visible to the analyzed malicious app.

**Bridging the semantic gap.** Given the security guarantees that approaches based on syscall-level analysis would get us, it is clear that, ideally, this approach should be preferred. In practice, however, the information they extract is too low level, making their results difficult to be interpreted. The conceptual problem is that, in the general case, it is challenging to recover the high-level *semantics* of an app's behavior solely starting from the list of recorded syscalls. For instance, even a simple operation, such as instantiating an SSL connection to a remote server, which

an Android app can perform by invoking a single high-level API, generates a complex sequence of multiple syscalls, most of them seemingly unrelated with the triggering of the high-level functionality. In this specific case, for instance, the complexity is due to the fact that the analyzed app ultimately has to invoke a series of syscalls belonging to different technical areas to complete this task, including inter-process communication with the system service that provides the trusted CA certificates, random-number generation for creating the nonce used to setup the connection, and network-related syscalls to perform the handshake with the remote server. While there are few works that reconstruct parts of this behavior (e.g., CopperDroid [TKFC15a] focuses on reconstructing the semantics of specific activities, including Binder-related operations), it is not clear whether reconstructing this semantics gap is in fact possible or practical in the *general* case. In fact, even though it may be practical to scan for specific patterns in a sequence of syscalls, traces often contain thousands of syscalls that do not seem to relate to any common pattern.

**Goal of this work.** To date, we are not aware of any work that actually investigates the feasibility of reconstructing the high-level semantics from generic low-level syscall traces. The goal of this work is to fill this gap: *this chapter presents the first systematic exploration of the challenges and feasibility of bridging the gap from trustworthy system calls to semantics-rich, but difficult-to-obtained high-level APIs.*

To this end, we have built a new analysis framework aiming at exploring the complexity of this research problem with a data-driven approach. The first key challenge is the scale: by dynamically analyzing 750 Android apps, we have collected data on over 40 million API invocations, which in turn generated over 13 milion syscalls invocation (interestingly, many API invocations do not invoke any syscall). We then process this low-level data to build a knowledge base of so-called "models," which aim at summarizing the big amount of raw data that we have collected in the previous step, to make it more viable for subsequent analysis. The complexity of the Android framework, the high number of exposed high-level APIs, the API's non-deterministic behavior, and the overlap generated by these APIs, make the analysis of this dataset far from trivial. To the best of our knowledge, this work performs the first data-driven exploration of this problem space, and it provides evidence that this is a very difficult problem, significantly more challenging than what previously thought.

In summary, this work brings the following contributions:

- We systematically explored the research problem of *semantically*

*lifting* a generic trace of performed system calls to a trace of invoked high-level APIs, with a focus on Android.

- We built a large-scale, annotated dataset that maps high-level APIs to the various "representations" of low-level syscall traces, and we provide an in-depth discussion of patterns and other interesting aspects.
- We develop and test different approaches attempting to perform the aforementioned semantic lift problem, and we show that this is a much more difficult problem than what previously thought.
- We provide recommendations and lessons learned that future work needs to consider when tackling this problem.

In the spirit of open research, we make our instrumentation framework, the collected dataset, and the analysis results publicly available at: https://github.com/eurecom-s3/syscall2api.

## 5.2 Background on Dynamic Analysis

**Android framework API.** Programming languages are commonly divided in two categories, depending on whether they provide a high- or a low-level abstraction over the computing system. High-level programming languages provide to the programmer a closer experience to a natural language and they are designed to perform complex tasks in few lines of code. On the other hand, low-level programming languages allow the programmer to interact with those aspects of computation that high-level programming takes for granted.

High-level programming languages expose to programmers a set of functionalities, called Application Programming Interface (API). In Android, these APIs are implemented in the so-called Android Framework. Some of these APIs are not implemented solely in Java, since their behavior exceeds the expressiveness of this language. They rely instead on the Java Native Interface (JNI), which provides a bridge toward parts of the framework written in C or C++. This is the case for some of the most complex and, arguably, the most security relevant APIs, like those that handle the personal data of the user, interact with the broadband, access the Internet, etc. Indeed, those functionalities require the intervention of the operating system to be accomplished. Being based on the Linux kernel,

in the Android operating system a user space application can take advantage of the services exposed by the kernel by means of system calls (syscall from now on).

Even though it is true that any security sensitive operation is performed by means of syscalls, the contrary is not true. In fact, a vast number of syscalls are actually invoked to implement behaviors that are not strictly security-sensitive, such as user interaction, memory management, and thread synchronization.

It is important to note that not only the framework, but also apps can contain pieces of code written in low-level languages. Moreover, both the high- and low-level code run in the same process and with the same privileges and there is no security boundary between the two. This is a common misconception, which led previous works to overlook the role of native code in the realm of Android dynamic analysis [ABF+16].

**Dynamic analysis.** Understanding the behavior of a program is an important step toward determining whether it is malicious or not. Dynamic analysis aims to gather this information from running the program in a controlled environment, recording as much evidence of malicious activities as possible.

Depending on the type of the controlled environment, the collected information can vary. Execution traces are one of the most common types of evidence collected during dynamic analysis and they describe a timeline of what was executed in the context of the program under analysis. Different granularities are possible, including API- and syscall-level traces.

API tracing records all the high-level functions invoked during the execution. Different mechanisms have been proposed to obtain such traces, including framework modification, run-time hooking and Ahead-of-Time (AOT) compilation instrumentation. Unfortunately, all of them can be detected and evaded by native code components.

Framework modifications, for example, can be identified by a malicious application through memory introspection. Moreover these techniques rely on the assumption that the program uses the default run-time provided by the system, but a malicious application could ship its own run-time library as a native library, avoiding completely the instrumentation. Run-time hooking and AOT compilation instrumentation suffer from similar problems. They both assume that the malicious code is implemented by the app in the high level language. However, the malicious behavior could be perpetrated by the native code, for example by mimicking the same syscalls that the framework would invoke to complete the same task. More fundamentally, the fallacy of API tracing mechanisms re-

sides in that the instrumentation is in the same security context of the program under analysis.

On the contrary, syscall traces can be obtained directly from the kernel, in a transparent way from the program perspective. There are several techniques to acquire syscall traces, the two most prominent being `strace`, a ptrace-based mechanism, and SystemTap [EH06], which inserts probes in kernel space and logs relevant information. The main drawback of syscall tracing is that the information collected are difficult to interpret. Finding evidence of malicious activity from a syscall trace alone can be a hard task.

## 5.3 Challenges

Reconstructing the semantic gap from a syscall trace is a task made particularly difficult by several challenges, which this section systematizes. We note that the discussion of these challenges is "conceptual" — a priori, it was not known whether these challenges would or would not actually pose problems when dealt with in practice. To the best of our knowledge, in fact, no previous work has ever explored the actual practicality issues that these challenges create. One of the contributions of this work is to fill this gap: as we will present throughout the chapter, our experiments provide the first data-backed evidence that these challenges do cause profound problems.

**Multiple possible execution paths.** The first challenge is that different invocations of the same API could follow different execution paths. This could be the case for a number of reasons. An API could behave differently depending on the arguments with which it has been invoked. However, its behavior could also differ depending on the execution environment and context. Different execution paths of course imply that the number and type of syscalls that are executed upon API invocation can widely vary. For example, consider an HTTP-related API: from the perspective of syscalls invocations, the recorded trace can widely change depending whether the API's argument is a valid URL, or whether the device has network connectivity. These aspects can clearly influence whether we would see network-related activity in the syscall traces.

**Non-determinism.** Another potential problem is non-determinism. With this term, we refer to those cases for which even if an API is invoked with the same arguments and within a "similar" environmental context, the syscall traces could still differ due to inherent non-determinism of the system or because of very subtle "internal" differences (e.g., the current internal state of the memory allocator). Naturally, one could argue that the

lowest-level aspect of the system could be considered as part of the "context" and that this challenge is overlapping with the previous one. This would be, of course, a valid argument. Nonetheless, we opted to make this distinction explicit due to the different nature of the source of potential divergent behaviors. As we will discuss throughout the chapter, the different nature greatly influences the *frequency* with which such non-determinism arise and *how* these problems should be tackled in practice.

**Multiple layers of APIs.** The Android framework is organized as a complex, multi-layer system of APIs: each API, especially the ones "exposed" to third-party apps, are implemented by invoking several others lower-level APIs. Indeed, it is rare that a high-level API directly invokes syscalls. This means that, when dealing with these higher-level APIs, every potential behavioral difference and non-determinism that affect lower-level APIs will be somehow combined—in a potentially combinatorial way. This makes capturing all different behaviors of a non-trivial API very challenging in the general case.

**Inherent ambiguity of syscall traces.** Different APIs often use the same syscalls to implement their behaviors. In other words, a given sequence of syscalls can (and often does) overlap across the execution of different APIs. From the perspective of analyzing a syscall trace to then understand which APIs have been actually invoked, this poses a significant challenge: it is very complex to "go back" with certainty as there are many different possibilities that may explain a particular sequence of syscalls. We then say that these syscall traces are *ambiguous* as it is often not possible to determine which, across a number of potential candidates, is the real API that has been actually invoked.

**No clear boundaries.** Given a syscall trace, it is challenging to determine when the syscall sub-trace of a specific API is starting or ending. In fact, there are no clear-cut markers signaling these aspects, and traces of different APIs (or even of the same one) may have different lengths. This aspect, together with the other aspects and the combinatorial nature of how low-level APIs are used to implement higher-level APIs, makes associating a series of syscalls to a given API much more challenging.

**Event-based nature of Android apps.** Android apps are written following an event-driven paradigm, which implies that apps often make use of callbacks. The classic example is the definition of an `onClick` callback to define what should happen when the user clicks on a specific button.

This pattern often causes several, nested control flow transitions from the Android framework to the Android app, and vice versa. In fact, con-

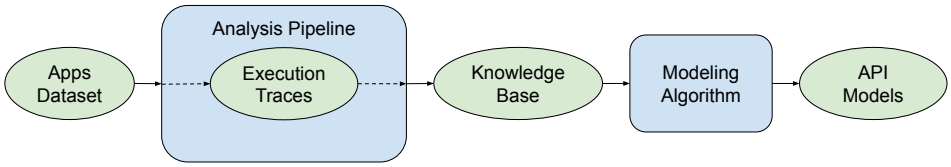


Figure 5.1: Overview of the approach.

sider what happens in the scenario where a user clicks on a button: 1) the control flow transitions from the framework to the `onClick` callback method, implemented in the app; 2) the `onClick` method may invoke several Android APIs, which would cause the control flow to transition back to the Android framework; 3) when the execution of these APIs is over, the control flow goes back to the `onClick` method; 4) when the `onClick` method ends its execution, the control flow goes back to the Android framework once again. These various control flow transitions make our analysis significantly more complicated. We note that these problems do not affect more traditional programs that do not heavily rely on asynchronous callbacks.

**APIs cannot be invoked without proper context.** With the aim of collecting data about which syscalls are invoked by which API, one possibility would be to consider each API separately and automatically invoke it within an instrumented environment. Unfortunately, this approach would not work in practice. In fact, the vast majority of APIs need to be invoked with the appropriate context, or otherwise they would quickly quit their execution due to errors. Moreover, many of these APIs require a proper "receiving" object to be invoked on, and the automatic creation of such objects is a very challenging task per-se.

## 5.4 Approach

This section discusses how we approached the various challenges discussed in the previous section. Our approach is summarized in Figure 5.1.

The first step of our approach consists in building a dataset containing *which syscalls are invoked by which API*. Conceptually, the idea is to use this dataset as a sort of ground truth, to then use it to perform additional experiments. As mentioned earlier, this task is challenging per-se. In fact, we cannot just create code to execute the various APIs, as we would not know with which arguments we would need to invoke them and from which context. We approached this problem by taking a large number of

```
API A: Entering
  Syscall w
  Syscall x
  API B: Entering
    Syscall y
  API B: Exiting
  API C: Entering
    Syscall z
  API C: Exiting
API A: Exiting
```

Listing 5.1: Example of an analysis trace.

*benign* Android apps and by executing them in an instrumented environment to produce both API- and syscall-level traces. This step is discussed in Section 5.5.

This raw data is sparse and contains a remarkable amount of redundancy, and the scale of this data does not make it suitable to be used for additional analysis without a pre-processing step. Thus, in a second step, the raw data is then organized in a data structure (that we refer to as *knowledge base*), which lays the foundation for subsequent analysis. For this step, the idea is to eliminate unneeded redundancy and to somehow obtain a concise representation of what contained in the dataset. The output of this analysis is a set of so-called *API models*. These models offer a "usable" over-approximation of all the behavior collected during the initial analysis phase (see Section 5.6).

The specifics of these API models have been designed to be useful for two different purposes. First, we perform the first empirical data exploration on this peculiar dataset (see Section 5.7), and we use it to uncover patterns and high-level metrics that show how challenging the problem of semantics reconstruction actually is. Second, we use these API models to take the first steps toward *mapping a generic sequence of syscalls to their associated APIs*, as discussed in Section 5.8.

## 5.5 Knowledge Base

In this section we present the methodology we followed to create our knowledge base. We started by considering a set of *benign* Android apps, which we then analyzed within our analysis framework, discussed in this

section. This analysis framework consists in an instrumented environment capable of logging traces of *both* syscalls and APIs. These raw analysis traces are then parsed and loaded in a more suitable tree-based data structure.

### 5.5.1 Analysis Tracing Pipeline

**Syscall-level tracing.** To log the syscalls invoked by a given app we relied on `strace`, which is a robust, off-the-shelf tool based on the `ptrace` syscall. For each syscall, we traced the timestamp, the calling *thread id*, the syscall name and its arguments. For obvious performance reasons, this behavior can be selectively enabled on the application under analysis only, so to avoid to slow the entire system down with unneeded instrumentation.

We note that, in principle, relying on strace has two disadvantages. First, it can be detected by an app. While this is true, this is not a problem at this stage because our goal is to collect the behavior of benign apps, which we assume to not contain anti-debugging techniques. Moreover, strace can be completely implemented in kernel space [HKFK18, EH06], making it more resilient to anti-debugging techniques and suitable for the analysis of malicious programs. Second, strace can cause a significant slowdown. However, once again, this is not a significant concern in our scenario as we are analyzing apps to collect "as much behavior as possible," and we do not necessary need to cover "all" the behavior of an app. In other words, while the slowdown may make us lose some behavior, this aspect does not threaten the validity of our experiments. The aspect that is actually of critical importance is that all the events (both syscalls and APIs) are logged in the appropriate chronological order, which is the case for our system.

**API-level tracing.** To log the APIs invoked by a given app, we first considered well known instrumentation frameworks, such as Xposed [Dev17] and Frida [fri]. In fact, one of the main features of these frameworks is the possibility of tracing specific API methods. Unfortunately, it turns out that when attempting to hook more than a few hundreds APIs, these frameworks make the system unstable, leading to repeated crashes. This is a problem as the Android framework is constituted by tens of thousands of APIs.

To this end, we have developed a new solution, which is based on source code instrumentation. By means of JavaParser [javb], we automatically instrumented *all* the public methods in the AOSP framework. In particular, we added a call to `logApi()` — a new static method that we defined in the `java.logging.Logger` class — at the entry point and at all exit points

(e.g., return statements, catch blocks of exceptions) of every instrumented method.

The `logApi` method takes a string as its first argument, which our instrumentation pass uses to specify *which* API has been invoked. In particular, this string contains the name of the instrumented method and whether the call originates from an entry point or an exit point (i.e., whether the method has been just invoked or whether its execution is about to end). Under the hood, this `logApi` method simply invokes a `write` syscall, using as an argument the same argument received by the `logApi` method itself.

This technical solution gives us a setting where both syscalls and APIs logging converge in the same *unified tracing channel*. Since these analysis traces are also thread-aware (by simply logging the thread id of the thread that invoked the API or syscall), all the log entries are already chronologically ordered and consistent, by design.

**Example of an analysis trace.** The result of this step is a merged analysis trace, which contains both syscalls and APIs, with the corresponding thread id and timestamp. Listing 5.1 shows an example of an analysis trace. In the listing, it is possible to see how the system can transparently log both *enter* and *exit* events for both syscalls and APIs.

### 5.5.2 Building a Knowledge Base

The analysis traces created in the previous step contain *all* the information collected, but they are not easily processable. To this end, we post-process these traces and we organize them in a more suitable data structure. This structure consists in a key-value store in which each key is the fully qualified method name of an API, and each value is a list of entries, each of which represents a specific instance of an API invocation. Each of these entries contains a list of events recorded between the start and the end of that specific instance of the API invocation. The events can either be “syscall invocation” or “API invocation.”

## 5.6 API Models

Invocations of the same API usually share common features but they are not always completely identical. For example, the two different branches of an if-else construct in the API code can lead to different sequences of API calls or syscalls (see Section 5.3 for a more systematic discussion of

similar challenges). The knowledge base discussed in the previous section contains all relevant information and it can be already used as a source of interesting data. However, it cannot be used to recover the high-level semantics without some kind of pre-processing. The reason for this is that APIs can potentially have a big number of different invocations (see Section 5.7) and each invocation can be significantly different from others.

In this section we introduce the concept of *API models*, their design and the rationale behind it. We then present an algorithm that creates API models starting from the invocations of an API. The last part of this section discusses how a sequence of syscalls can be then matched against these API models.

### 5.6.1 Anatomy of an API Model

In the context of this work, an API model is an object that summarizes the common features between different invocations of the same API. A model is constituted by an ordered sequence of symbols representing the various APIs and syscalls found in the invocations. Each symbol in the model can have an optional *modifier* that indicates that it can appear up to an unlimited number times.

API models represent an over-approximation of all the information stored in the knowledge base. In fact, by assuming that a syscall pattern can be repeated up to an unlimited number of times, an API model can match sequences that have not been observed in the API invocations.

The choice to model repetitions of syscalls in this way is driven by the intuition that repeated patterns generate from loops in the execution. Those loops can be repeated either a fixed or a variable number of times. Our API models make use of the repetition modifier only if the same pattern has been observed repeating itself a different number of times in distinct invocations.

### 5.6.2 API Models Creation Algorithm

The model creation phase consists in applying a two-step algorithm to all the APIs in the knowledge base. In the first step we identify all those invocations that are identical according to the following definition: *two invocations are identical if they contain the same API calls and syscalls in the same order*. Note that the syscalls arguments are not taken into account. Duplicate invocations are not taken into account from further processing.

In the second step the algorithm processes each of the remaining invocations to create a list of models for each API. In particular, the algorithm proceeds as follows. It first attempts to find the longest repeated pattern in the invocation under analysis. If a repeating pattern is found (e.g., a single syscall or a sequence of syscalls that keep repeating itself), the algorithm creates a model similar to the original invocation, except for the repeating pattern that is marked with the repetition modifier. This model is then checked against the other invocations. If at least one of them produces a match, the generalization is considered "useful" and the model is added to the list of API models. If not, it means that this over-generalized model was not useful: the algorithm thus discards it, and adds to the list of API models the trivial model matching the "exact" invocation under analysis.

### 5.6.3 API Models Matching

Once these API models are computed, the next step is to approach the *mapping problem*. Given a sequence of syscalls, this problem aims at determining which API is the most likely to have generated such a sequence. In a way, the goal is to *map* a given sequence to the "correct" API. There are of course many different possible algorithms and strategies to implement this.

For this work, we opted to implement two *extreme* strategies: the longest match and shortest match strategies. In both cases, the algorithm starts from the very beginning of the syscall sequence. It then considers all the API models in our database and it determines which of these API models actually match the given sequence of syscalls. The algorithm then selects the longest (or the shortest) of these matches, and this initial sequence of syscalls is considered as covered. The algorithm then proceeds by applying the same method to the sequence of syscalls that followed the one that is covered by the selected API model.

We note that this constitutes the first step into reconstructing the API trace starting from a sequence of syscalls. We present an evaluation of these two strategies in Section 5.7. Of course, we acknowledge that there are in fact many other potential strategies. However, we believe that considering these two extreme strategies is a promising first step toward understanding and exploring this relevant research problem.

## 5.7 Data Exploration

This section explores our knowledge base (KB) by discussing interesting statistics and insights. We start by giving more precise information about the apps that we analyzed for collecting the analysis traces and the experimental setup. We then present measurements about the information stored in the KB. These measures provide empirical data highlighting the difficulties in reconstructing the high-level semantics from generic low-level syscall traces. Specifically, we will show how the APIs in the KB are very diverse and, because of their nature, how different APIs present different challenges for semantic reconstruction. We will also show how a human analyst can query the KB and how this is important in highlighting the problematic nature of two aspects, namely noise and ambiguity, making semantic reconstruction a task more challenging than what previously thought. We then explore these two aspects in an automated fashion and we discuss the gained insights in Section 5.7.3 and 5.7.4, respectively.

### 5.7.1 Apps Dataset and Experimental Setup

As mentioned throughout this chapter, we opted for a data-driven approach to explore the problem of semantics gap reconstruction. We build our ground truth of analysis traces by recording the execution of a set of 750 apps. We collected these apps from the F-Droid Open Source Android App Repository [fdr]. In particular, we selected *all* apps from this dataset that used at least one dangerous permission. The rationale behind this choice is that these apps would tend to be more complex than others not requiring any permission, and would thus have more chances to expose interesting behavior.

Each app was executed for five minutes on a Google Nexus 5X device, which was previously instrumented with our modified Android framework, as described in Section 5.5.1. We then used the Android Monkey Runner [Goo] to stimulate the app's user interface. We fully acknowledge that the Monkey Runner is not sophisticated and may not trigger deep parts of the app's codebase. However, we note that the rationale behind these experiments is not to fully cover a specific app, but to execute many apps and collect what we can from each of them.

### 5.7.2 API Classification and Statistics

Our KB contains invocations for a total number of 4,630 distinct APIs. The total number of API invocations observed is over 40 million, while the to-

| | Leaf APIs | Non-Leaf APIs |
|---|---|---|
| **Empty APIs** | 1730 | - |
| **Monoform APIs** | 29 | 810 |
| **Multiform APIs** | 573 | 1488 |

Table 5.1: API occurrences in KB. Note: there cannot be Empty APIs that are also Non-Leaf APIs

| | Leaf APIs | Non-Leaf APIs |
|---|---|---|
| **Empty APIs** | 2850 | - |
| **Monoform APIs** | 59 | 665 |
| **Multiform APIs** | 94 | 962 |

Table 5.2: API occurrences in KB (after noise reduction)

tal number of syscall that these API invocations invoked is over 13 million. Note that the number of API invocations is larger than the number of syscalls, which means that a significant number of API invocations do not result in any syscall. In average, each API has been observed 8,721 times, while the average number of events in the invocation lists is 0.84 (3.63 without considering empty invocation lists).

We categorize the APIs according to two different aspects. First, we consider how many different entries each API has in its invocation list. That is, for each API we look at how many different syscall sequences we have recorded in our dataset. We distinguish three different cases: 1) *Empty APIs*, those whose invocations are all empty lists; 2) *Monoform APIs*, those for which all the invocations are equals (and non-empty); and 3) *Multiform APIs*, those that have at least two different non-empty invocations. Second, we cluster APIs according to the *type* of events that each of their invocations contains. For this aspect, we distinguish between 1) *Leaf APIs*, those whose invocations contain only syscalls, and 2) *Non-Leaf APIs*, those containing at least one API in their invocation lists.

Table 5.1 shows the occurrences of each category of APIs in our KB. It is interesting to understand how each category of API plays a different role in the context of semantic reconstruction. *Empty APIs* are those that are completely implemented in user space and do not make any system calls. For this reason, their behavior cannot be identified at all based on syscall information only. *Multiform APIs* make the overall task of semantic reconstruction very challenging because all of their invocations must be taken into account. Our modeling algorithm, for example, could produce more

than one model for each API in this class. *Monoform APIs*, on the other hand, are simpler because their only invocation can also be used as model.

*Leaf APIs* are the closest to syscalls in terms of semantics. Some of them invoke always the same syscall (e.g., *android.net.LocalSocket.setSoTimeout* always executes the syscall *setsockopt*). They are also the easiest to reconstruct, since their behavior can be recognized directly from the executed syscall. To reconstruct a *Non-Leaf API*, instead, one needs first to reconstruct the other APIs that it could invoke.

### 5.7.3 Noise Patterns Identification

While inspecting the data offered by our knowledge base, we came across surprising insights. For example, we found some syscalls in the invocation list of a few of those APIs that were expected to be *empty*. One example is the *java.lang.StringBuilder.append* API. Interestingly, while the vast majority of the invocations of these APIs were indeed empty, (very) few of these invocations contained peculiar syscall patterns that, at first glance, we could not explain with the expected behavior of the API. To our surprise, we then found that these patterns were also observed in the models built for *other* APIs.

To investigate this unexpected finding, we developed a post-process analysis pass to automatically identify similar cases. The key idea is to perform anomaly detection. Our system identifies a model of an API as an outlier if the model describes a number of invocations (of that API) that is lower than a certain threshold (for our tests, we used $1/1000$ as a threshold). With this tool, we identified three "noise" patterns. The first relates to thread synchronization (e.g., *futex*, *sched_yield* and *clock_gettime* syscalls). The second relates to memory management (e.g., *madvise* and *mprotect*) or their combinations. Finally, the third pattern we identified relates to the specific malloc implementation in Android's *bionic* C library: since it is used to obtain memory for the allocation of new objects, it can potentially appear during the invocation of any API that allocates Java objects—and, similarly to the other two cases, this is what causes the noisy pattern.

To better explore the role that these noisy patterns have in our dataset, we opted to eliminate from our models all the syscall patterns that contribute to such noise (i.e., the ones mentioned above), since we believe that these classes of syscalls do not carry any meaningful information that can be used to reconstruct any API semantic. Table 5.2 shows statistics

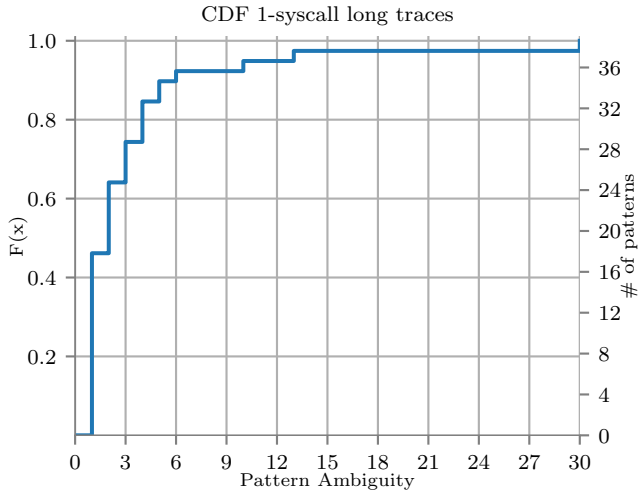


Figure 5.2: Pattern Ambiguity for 1-syscall long patterns

for each class of APIs in the KB after removing the noisy patterns. A comparison of these data with those in Table 5.1 suggests that noise reduction leads to more consistent data. For instance, the number of APIs that have two or more different invocations decreased by almost 50%.

### 5.7.4 Ambiguity Measurement

Another aspect we explored relates to the inherent ambiguity of models included in our KB. For example, we noticed that some models overlap or are identical, even though they belong to different APIs. This means that potentially more than one API model can provide a match for the same syscall pattern, leading to ambiguity in the results of any matching algorithm.

With the goal of quantifying the ambiguity of the results in our dataset, we define a new metric, which we call *ambiguity score*. This metric is an integer number that can be computed for *each pattern of syscalls*. We define this metric as the number of different APIs (in our KB) that match against a given pattern of syscalls. We tackle this problem by considering syscalls patterns of different lengths. Moreover, we consider two different values: *pattern ambiguity score* and *total pattern ambiguity score*, the only difference between the two being that in the latter case we weight a pattern according to how many times it appeared in our traces. Figure 5.2 and Figure 5.3 show the cumulative distribution functions (CDF) of the ambigu-

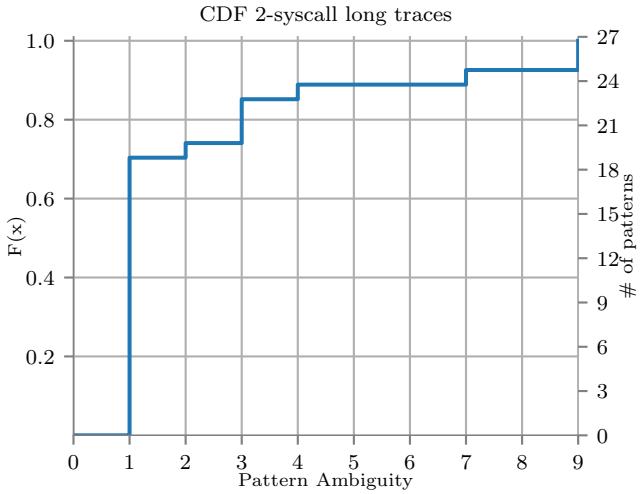


Figure 5.3: Pattern Ambiguity for 2-syscall long patterns

| | **Shortest Match** | | **Longest Match** | |
|---|---|---|---|---|
| | w/ noise | w/o noise | w/ noise | w/o noise |
| **Trace coverage** | 26.4% | 45.2% | 33.0% | 61.5% |
| **Correct matches** | 30.9% | 38.9% | 26.4% | 46.9% |

Table 5.3: Results of two variants of the matching algorithm.

ity score for 1-syscall and 2-syscall long patterns respectively (after having removed the noise). Figure 5.4 and Figure 5.5, instead, plot the CDF of the *total* ambiguity score, for the same patterns. These figures show the data before and after removing the noise. For a better visual comparison between the CDFs, the figures show the data around the point in which the CDF of the noiseless KB reaches 1.0. The CDF of the noisy KB instead reaches 1.0 at much higher abscissa (not shown in the figures), since it raises at a slower pace with respect to those of the noiseless KB. This is also true for the CDFs built for other syscall pattern lengths, meaning that models built without removing the noise are more ambiguous than their noiseless counterpart. Figures 5.6 to 5.11 show the CDFs for patterns of different lengths (from three to up to five).

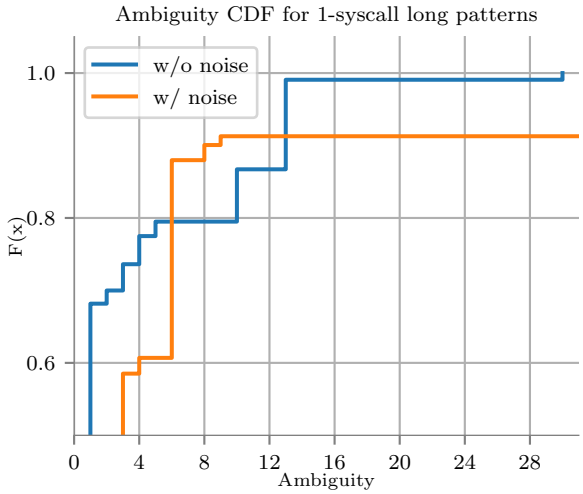


Figure 5.4: Total Pattern Ambiguity Comparison (1-syscall long patterns)

| | Method | Class | Package |
|---|---|---|---|
| **Trace Coverage** | 61.8% | 62.0% | 63.1% |
| **Correct Matches** | 46.9% | 47.0% | 49.3% |

Table 5.4: Accuracy results under relaxed definition of correctness.

## 5.8 Exploring the Mapping Problem

This section discusses a first attempt to reconstruct the semantics gap of a *generic* sequence of syscalls. The input to this step is a non-annotated syscall trace and we investigate how two strategies would perform in this context. The challenges discussed in Section 5.3 make this task particularly difficult.

To this end, we define the notion of correct match as follows. A match is *correct* if it spans the same syscalls of an API in the annotated trace and if said API is in the set of those that the algorithm selected as candidates. This means that a match is not considered correct if, for example, it covers more syscalls than the ones actually produced by the API, or if it does not start exactly on its first syscall.

We measure the results of the reconstruction process in terms of percentage of APIs correctly identified (i.e., the ratio between the number of APIs correctly identified and the total number of APIs in the annotated trace) and percentage of the traces covered by correct matches (i.e., the ra-

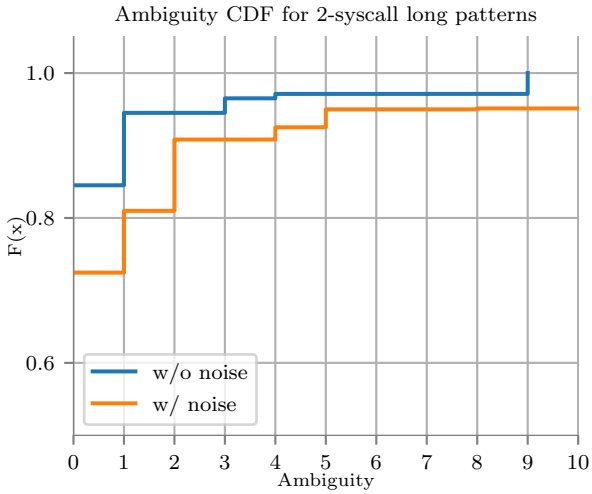


Figure 5.5: Total Pattern Ambiguity Comparison (2-syscall long patterns)

tio between the number of syscalls correctly assigned to a candidate API and the number of syscalls effectively in the trace).

We implemented two variants of a matching algorithm: a *shortest match* and a *longest match* policy. Table 5.3 reports the results in terms of *trace coverage* (i.e., which percentage of the trace was possible to cover) and *correct matches* (i.e., the ratio of matches that are correct). These results show that, clearly, the "longest match" heuristic results are better, as it produces higher rates of correct matches and trace coverage in each single test. Table 5.3 also provides a comparison of the results obtained by adopting the models built before and after noise reduction. It is interesting to note that not only noise reduction decreases the ambiguity (as shown in Section 5.7.3), but it also increases the amount of correct matches.

We note that the results reported thus far use a quite aggressive definition of "correctness." In a way, we consider a match correct if and only if the algorithm is able to identify the specific, exact API. Since there are cases in which different APIs belonging to the same class (or package) actually have the same semantics, we decided to explore how the accuracy would change under a more relaxed definition of "correct match." Table 5.4 shows the percentage of trace coverage and match correctness when a match is considered as correct in three different scenarios (using the "longest match" heuristics and after noise removal): we consider a match correct if the method name, the class name, or the package name of the API matched (instead of its fully qualified name, which also contains

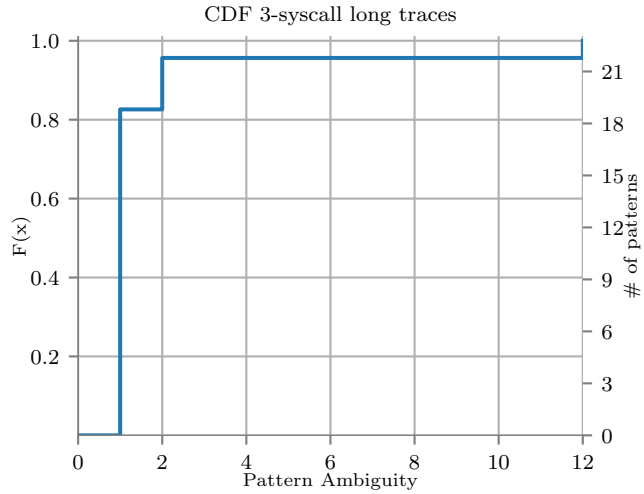


Figure 5.6: Pattern Ambiguity for 3-syscall long patterns

the types of its arguments). The table shows that the numbers do improve, but that they are still far from ideal. We believe that the reason for the low accuracy of the results of our algorithm resides in the fact that it fails in recovering from an incorrect match caused by a length mismatch. In this situation, the algorithm is out of synchronization as it tries to match the next API from the very next syscall in the trace.

Another source of desynchronization is given by those sections of the trace in which no APIs are recorded (i.e., when the execution of the application returns to the framework, see Section 5.3 for more details). This issue cannot be solved applying the same approach used for APIs, but they do not share enough features to build meaningful models. Moreover, these areas contain similar patterns to those observed in API models, making it difficult to distinguish between them.

**Discussion.** These results show that simple strategies are far from enough to properly map sequences of low-level syscalls to high-level APIs. The mapping task appears even more challenging when considering that our experimental setup made the analysis, in theory, much simpler than what it would be in a real scenario. In fact, our experiments attempt to map sequences of syscalls to APIs that have been extracted from the same knowledge base. In a real scenario, instead, the algorithm would not have any guarantee that the potential target API is one API already in the knowledge base (as an unknown app may make use of APIs never seen in the train-

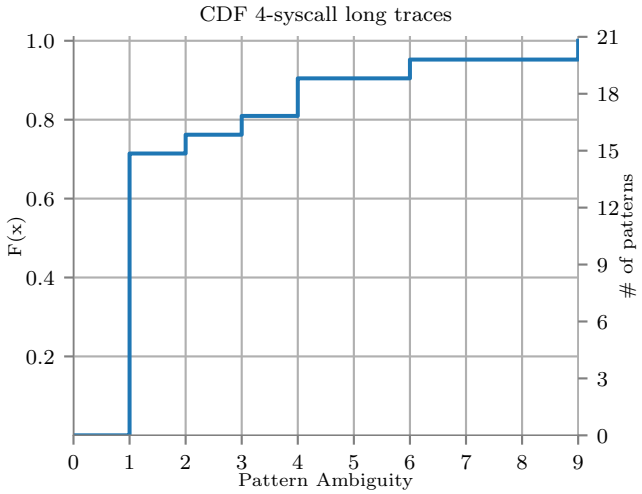


Figure 5.7: Pattern Ambiguity for 4-syscall long patterns

ing set). Moreover, we even simplified the problem by assuming that the starting point of the first API in the trace is known. Last, our "correctness" definition is quite generous, as it considers an API match as correct if the correct API is among one of the selected candidates: ideally, the perfect matching system should indicate only one API for each match. We believe these observations strength our key hypothesis: *that, even under these very favorable conditions, the mapping problem presents inherent difficulties that are very challenging to overcome.*

**Future directions.** We believe that reducing the noise in the models is a key component for a successful approach to the mapping problem, since we identified that the noisy patterns produced by the framework and the ART runtime a strong source of ambiguity. To this aim, we believe that future work should investigate a more aggressive noise reduction strategy to improve the results. Another direction for improvements could be to leverage the fact that some APIs are often used in conjunction with others, providing a heuristic for choosing between multiple candidate APIs.

At the moment, our system collected information about the APIs exposed via Java public method. A different approach would be to monitor the invocation of every single Java method (including private ones) in the framework. On the one hand, this approach could shed light on those areas of the traces that seemingly do not contain any API. On the other hand, private methods are more difficult to interpret for an analyst since they

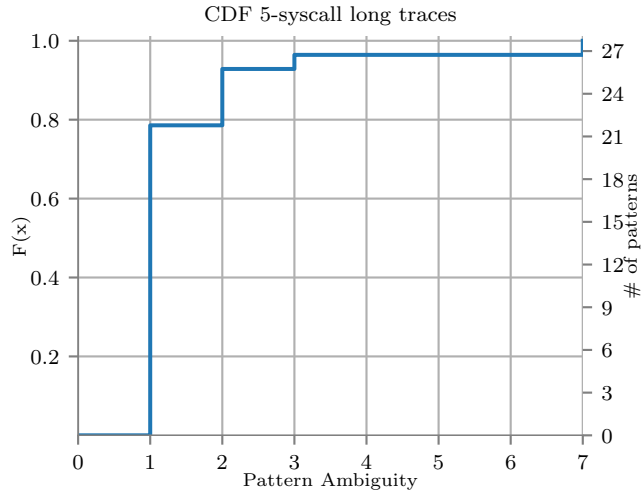


Figure 5.8: Pattern Ambiguity for 5-syscall long patterns

are undocumented and require knowledge of the internals of the Android framework to be fully understood.

Another step of our pipeline that can be enhanced is the model creation algorithm. In general, we believe that future works should focus on creating API models that describe as many features of the API invocations as possible. In this work we show how to model repeating patterns, but there are other features that are worth modeling. For example, one possible improvement can be to summarize in the same model all those invocations that differ for a small number of syscalls and/or API calls only.

In an attempt to model this type of scenarios, a first version of our prototype relied on the Needleman-Wunsch sequence alignment algorithm [NW70]. The rationale was that by aligning two invocations is possible to find those syscalls and API calls that appear in only one of them. We leveraged the aligned sequences to create models in which the symbols corresponding to these API calls and syscalls were marked with an additional modifier. This modifier indicates that the symbol to which it is applied is "optional," meaning that the model matches a sequence regardless of its presence. This approach, however, led to unacceptably high computational complexity of the pattern matching phase, to the point of making it unfeasible in practice for models with many entries. Still, we believe there could be some value in adopting this technique for only a subset of the APIs, especially those whose invocations contain only a small number of syscalls and API calls.

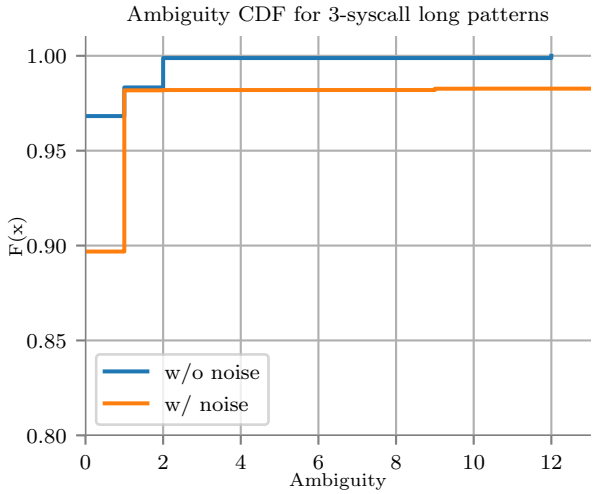


Figure 5.9: Total Pattern Ambiguity Comparison for 3-syscall long patterns

Lastly, we believe our approach would clearly benefit from integrating the results from existing systems like CopperDroid [TKFC15a], which already perform several steps to recover the semantics of the *ioctl* syscalls. This particular class of syscalls is among the most frequent and ambiguous ones in our knowledge base. This is due to its fundamental role in the low-level implementation of the Binder subsystem, which relies on the *ioctl* syscall to exchange "parcels" of data between user apps and system services. Integrating CopperDroid with our system would add the corresponding Binder semantics to each *ioctl* syscall, which would eventually reduce the ambiguity for this class of syscalls significantly.

## 5.9 Related Work

The Android security community has published a vast number of works related to program analysis of unknown apps. This section places our work in the context of two main related areas, namely static and dynamic program analysis.

Several static analysis approaches have been proposed to analyze Android apps, and malware in particular. Some of the early works in this area include RiskRanker [GZZ+12] and DroidRanger [ZWZJ12], which rely on symbolic execution and a set of heuristics to detect unknown malicious applications. Another work is Apposcopy, which uses a signature-based

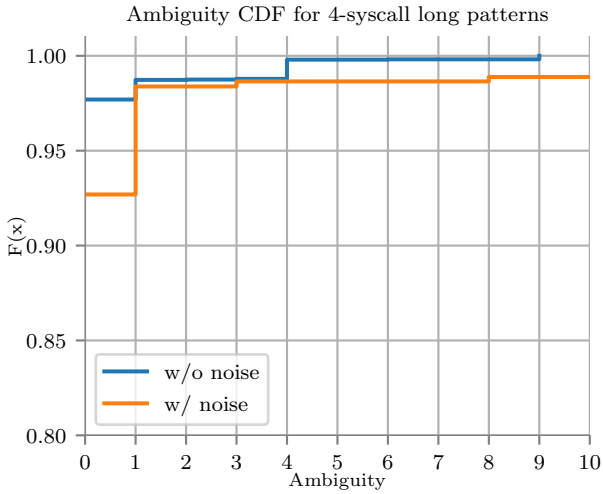


Figure 5.10: Total Pattern Ambiguity Comparison for 4-syscall long patterns

approach to detect known malware samples [FADA14]. Other works do not only focus on malware detection, but are more generic and attempt to identify suspicious data flows via taint analysis. Two relevant works in this area are FlowDroid [ARF+14] and DroidSafe [GKP+15].

Another important trend of works attempts to perform malware classification by using machine learning techniques. Some of the early works include Drebin [ASM+14] and DroidAPIMiner [ADY13], which both extract several features from Android applications (e.g., requested permissions, invoked framework APIs) and then apply machine learning techniques to perform classification. A different system is AppContext [YXA+15], which uses machine learning techniques to identify malware by using the "context" of each behavior as a feature. More recently, Mariconti et al. proposed MaMaDroid [MOA+17], a tool that uses Hidden Markov Model chains and, once again, starts from the API function calls to build behavioral models. Another recent work in a similar direction is SLAP [MRG+18], which also uses machine learning with features based on API-related information, with the difference that it attempts to be more resilient to adversarial samples.

There has also been extensive research on program analysis of Android apps through dynamic analysis. Enck et al. [EGC+10] present TaintDroid, a dynamic taint analysis that performs whole-system data flow tracking through modifications to the underlying Android framework and

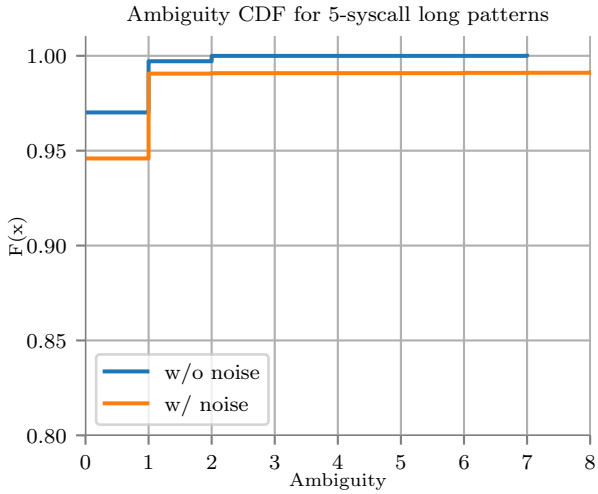


Figure 5.11: Total Pattern Ambiguity Comparison for 5-syscall long patterns

native libraries. Other efforts, such as Mobile Sandbox [SFE+13] and Andrubis [LNW+14], developed tools and techniques to dynamically analyze unknown Android applications. Another trend of works has proposed approaches based on dynamic analysis to perform multipath execution and dynamic symbolic execution on Java and Android applications [jpb, GKS05, MMP+12, WL16, RAMB16]. These approaches achieve higher code coverage than simpler dynamic analysis tools.

We note that *all these existing works use API-level information as the main building block for their analysis.* Their main rationale is that API-level information provides semantics-rich data, which in many cases is enough to discern benign apps from the malicious ones. This common trait of all these recent works underline the importance that API-related, semantics-rich data can play within the Android security research community. However, as often mentioned in this chapter, all these approaches can be detected and evaded [ABF+16].

One of the very few works that fully acknowledge this limitation and performs a step forward is CopperDroid [RFC13, TKFC15a]. In this work, the authors show how it is possible to reconstruct two categories of high-level behaviors. The first one consists in those implemented through Android Services, which CopperDroid identifies by unmarshaling objects used in Binder transactions (e.g., access to geolocalisation). The second includes those behaviors that result in a sequence of syscalls with a clear

data dependency, which CopperDroid reconstructs by means of a value-based data flow analysis technique (e.g., opening a file and performing operations on it).

However, the authors of CopperDroid also note that API-level information, while useful in reconstructing high-level behaviors in some cases, is not fully trustworthy when dealing with some complex scenarios. Take, as an example, the behavior of the *createSocket* method of the *SSLSocketFactory* Java class. This API creates an SSL tunnel over an already opened TCP socket. Exploring our dataset we noticed that to perform this task the framework invokes various syscalls, for example, "getrandom" to generate the nonce used for the encryption or "read/write" to perform the handshake. By simply inferring the data dependency between syscalls, CopperDroid would be able to recognize its network-related aspect, but it would fail in understanding that the syscall pattern is actually implementing a tunnelling mechanism that, according to the API documentation [java], enables to instantiate an SSL connection over a proxy.

Our work thus differs from CopperDroid by exploring the behavior reconstruction problem in a more generic way: given a list of syscalls, is it possible to build a pattern identification and "go back" from syscall to API in the general case? In other words, in this work we are interested in answering a more generic question, and we do not rely on specific patterns or on data-dependency among syscalls to address the mapping problem. The main difference with previous works is that we focused on using a data-driven approach to explore the more-generic problem of performing semantics reconstruction of a generic sequence of syscalls. In the process, we have also built the first dataset of API-syscall relationship — to the best of our knowledge, the first of its kind.

The problem of reconstructing high-level behaviors from low-level features is not exclusive to the Android security research field. Martignoni et al. [MSF+08], for example, modeled a set of malicious behaviors found in different malware families for the Windows operating system. To this aim the authors manually analyzed the executions traces of malware and benign programs to express high-level behaviors in terms of lower-level syscall-like events.

The main drawback of their approach is that the modeling phase cannot be automated because it requires human understanding of each high-level behavior. In our work, instead, the modeling phase is completely automated. Moreover, our approach is more generic since it models more than just a restricted set of behaviors and is not bound to malicious behaviors only.

# Chapter 6

# Future Work and Conclusion

## 6.1 Future Work

The research presented in this thesis suggests several avenues for future explorations in the field of malware analysis. In particular, we can highlight two strategies to build upon our work.

The first strategy consists of applying the techniques described in Chapter 3 to new domains. An obvious extension of our work on the PE ecosystem would entail analyzing other executable file formats. Our language, in fact, is generic enough to support other formats as well. This allows reusing our analysis framework as is, without any modification at all. We suspect that the software ecosystems built around other formats such as *ELF* and *Mach-O* are also affected by the problem of discrepancies, paving the way to evasion techniques that are yet to be explored systematically.

The case of *ELF* is particularly fascinating. Since this format is the defacto standard for most UNIX-like operating systems, a comprehensive analysis needs to consider the possibility that malware samples may attempt to camouflage as programs for another operating system, hiding their actual targets.

Our modeling language and differential analysis framework could also be used to study discrepancies in software handling other types of file formats. A particularly promising target may be the *PDF* format, which is known to be very complex and difficult to parse correctly [ELM16]. Malware commonly uses PDF files as vectors for infection by embedding malicious JavaScript code, which endpoint protection tools try to counter by extracting and analyzing these payloads. Previous works [CHY+16] have shown how malware exploits discrepancies in the extraction process to evade analysis. Future works may adopt and adapt the tools and methodology presented in this thesis to study the PDF parsing landscape systematically.

The file type inference system is another attack surface in antiviruses that the literature has found vulnerable to parsing differential attacks. Indeed, previous works have discovered that some antiviruses could not correctly infer the format of input files, opening the doors to evasion. Indeed, malware can confuse the antivirus into scanning it with signatures that reason about the semantic of the wrong file format. Jana et al. [JS12] provide a notable example, finding several of such exploitation techniques employing of blackbox fuzzing. We believe that this approach may have just scratched the surface of the type inference problem and that our techniques may actually help find more sources of discrepancies.

Lastly, our analysis framework can be used to generate valid seeds for fuzzing parser for file formats. In particular, one can leverage the *Corner Case Generation* technique described in Chapter 3.5.3 to produce a corpus of valid test cases for the software to test. By construction, these test cases trigger different paths in the original software, providing a good initial code coverage.

The second strategy aims at improving our work on the reconstruction of high-level semantics from low-level sources. In particular, an interesting way to tackle this problem could be to apply machine learning or pattern recognition techniques.

Another possibility could be to consider more information other than the sole syscall name, such as syscall arguments, invocation timing, and even some sort of data flow analysis among different syscalls. While our work on the semantic reconstruction ended in a negative result, there is no reason to believe that it cannot be solved, at least in some cases.

Moreover, while we focused on the Windows and Android operating systems, other platforms could also be studied, notably macOS and Linux. Both of these operating systems pose unique challenges. For example, while Windows and Android mainly support one set of APIs (the WinAPI and the Android Java API), Linux does not have one single programming language (thus, one single API set). This means that to recover high-level runtime information of a Linux program from its syscalls, one needs first to study the programming language in which the program is written and its APIs.

Finally, while in both our explorations of the semantic gap we dynamically analyzed non-malicious software, another possible future work could analyze malware, investigating whether it employs different APIs, or if the same APIs results in different syscalls if employed for malicious purposes (e.g., when malware invokes the same APIs, but with different arguments).

## 6.2 Conclusion

Due to its adversarial nature, the field of malware analysis and detection has become nuanced and very complex to master. Every wrong assumption in the design of anti-malware tools can open the door for evasion techniques that malicious actors do not wait to exploit to secure higher revenues for their harmful campaigns.

This thesis challenges two technical assumptions that the security industry has overlooked. For both of them, we provided systematic ways for exploitation, as well as guidelines and first approaches to solving the problems at their roots.

In particular, Chapter 3 presents our exploration of the parsing differential problem among software that handles the PE file format. Our methodology allows attackers and defenders alike to find discrepancies that malware can leverage to circumvent analysis and detection. This work suggests that the way security tools have handled executable file formats so far is conceptually flawed. In particular, security tools assume that each version of the operating system parses and loads the same executable in the same way. The results of our work completely overturn this conception by showing that implementations of the loader component of different versions handle edge cases differently. We argue that the next generation of security tools should treasure these findings and switch to a more sound approach to program loading.

Chapters 4 describes our work on API tracing bypass. Our novel technique creates self-contained executables that do not need any library provided by the Windows operating system. From a malware analysis point of view, such programs do not execute any APIs, effectively evading any instrumentation that relies on API-level information. In other words, this work proves that obtaining high-semantic information about a program's execution directly and in a reliable and unavoidable way is impossible.

This led us to study the feasibility of recover high-level semantic from low-level information sources, captured through techniques that (regular) malware cannot evade. In particular, in Chapters 4 and 5, we explore the semantic reconstruction problem from the syscall-layer to the API-level for the Windows and Android operating systems.

On top of the technical contributions highlighted throughout the dissertation, we believe that, on a more fundamental level, the merit of this thesis is that of trying to anticipate future trends in malware. Studying and questioning common practices and beliefs led us to discover new avenues for evasion.

By documenting our research work and its results, we hope to raise awareness among the security community about the shortcoming of our current techniques and possible new threats. To paraphrase Sun Tzu's "The Art of the war," only by knowing yourself and your enemy you need not fear the result of a hundred battles.

# Appendices

# Appendix A

# Loader Modeling

## A.1 Example of Constraints Model

```
1  INPUT foo 4
2  bar <- ADD foo 1
3  V1: ULE bar 10 term
4  V2: UGE foo 4
5  V3(V2): UGE foo 7 term
```

Listing A.1: Example of a model written in our language.

Listing A.1 provides a concrete example of a model written in our language. We now discuss the semantics of this model, and we will use this example as reference when explaining how other parts of our framework work.

At Line 1, the model specifies the existence of a 4-byte long input, named `foo`. At Line 2, a new symbol is introduced, `bar`, and the model specifies that it is *defined* as `foo + 1`. Note that multi bytes are parsed to integers as big-endian. Thus, adding 1 to a 4-byte field means adding 1 to its 4th byte, with carry. At Line 3, the model defines a boolean predicate, `V1`. The boolean predicate `V1` evaluates to true if and only if `bar` is less or equal than 10 (the "U" in ULE indicates it is an unsigned comparison). Moreover, the `term` keyword specifies that `V1` is a *terminal predicate*. This implies that, for an input file to be considered compliant, `bar`'s value *must* be less or equal than 10. At Line 4, the model defines another boolean predicate, `V2`. This predicate evaluates to true if and only if `foo` is greater or equal than 4 (once again, the comparison is unsigned). Note that, differently from `V1`, `V2` is *not* a terminal predicate. This means that, per se, whether `V2` evaluates to true is not a necessary condition for a given input file to be considered valid. The truth value of `V2`, however, is relevant for the conditional predicate `V3` specified at Line 5. The semantics of line 5 is the following: if `V2` evaluates to false, then `V3` evaluates to true, independently from the input value; if, however, `V2` evaluates to true, the specified predicate (i.e., *foo greater or equal than 7*) *must* also evaluate to true for `V3` to be satisfied. In addition to that, note that `V3` is a terminal predicate, which indicates that it *must* evaluate to true for an input file to satisfy the model. The net effect of this model is to constrain the value of the input in the ranges [0, 3] and [7, 9].

## A.2 Example of Translation in SMT problem

For sake of clarity, we now discuss how the example model discussed in the previous section in Listing A.1 is translated into an SMT problem.

The symbol definition at line 2 introduces the following formula:

$$bar \leftarrow foo + 1$$

Line 3 introduces the following predicate:

$$P_1 : \quad foo + 1 \leq 10$$

Line 4 is translated into the following predicate:

$$Q_1 : \quad foo \geq 4$$

The conditional predicate is instead translated as

$$P_2 : \quad Q_1 \Rightarrow (foo \geq 7)$$

The final formula of the SMT problem is then computed as the logic conjunction of all terminal predicates, in this case $P_1$ and $P_2$:

$$F : \quad (foo + 1 \leq 10) \wedge (\neg(foo \geq 4) \vee (foo \geq 7))$$

Thus, the SMT solver will be tasked to find an $foo$ such that the final constraint $F$ is satisfied. In this case, the constraint *is* satisfiable, and the SMT solver would return an integer value in the ranges [0, 3] and [7, 9].

## A.3 Excerpts from the Models of the Windows Loader

Listing A.2 shows the portion of the model of the loader of Windows 10 that handles the *Base Relocation* data directory.

Lines 2 to 4 parse the *RVA* and size of the relocation table. Line 7 introduces the boolean predicate `V6`, which evaluates to `true` when the relocation directory *RVA* and size are valid. All the following statements have `V6` as a precondition since they only make sense if the executable has a relocation table.

The logic that models the relocation table's content is embedded in the loop `L1` introduced at line 11 by the `VLOOP` operand. At each iteration of the loop, the variable `relocBlockAddr` contains the offset of the current relocation block in the file, starting from the value stored in `loopStart`. The boolean predicate `V99` (defined at line 21) is evaluated at the end of each loop cycle to determine whether the loop must continue. If that is the case, `relocBlockAddr` will be updated with the value stored in the variable `nextBlockAddr`. The last parameter of the `VLOOP` operand is the unroll

count that indicates the maximum number of times the SMT solver must consider the statements in the loop.

Each relocation block is followed by a number of relocation entries which depends on the block's size. The loop `L2` handles each relocation entry for the current block. For the sake of brevity, we will not describe the parameters of the `L2` loop, as they are similar to the one of `L1`.

The terminal predicate at line 34 determines the types of relocations that the loader supports. For Windows 10, this predicate evaluates to true if the relocation type is either 10, or less or equal to 4.

```
### Relocations
P: relocDir <- optHdr.DataDirectory[40, 8] as
    _IMAGE_DATA_DIRECTORY
P: relocVA <- relocDir.VirtualAddress
P: relocSize <- relocDir.Size

#### From ntdll!LdrRelocateImageWithBias:45,48
V6: AND (UGE optHdr.NumberOfRvaAndSizes 6) AND (NEq relocVA 0)
    (NEq relocSize 0)

P: tmpSize <- relocSize
P(V6): loopStart <- relocVA
L1(V6): relocBlockAddr <- VLOOP(loopStart, nextBlockAddr, V99,
    10)
    P: relocBlock <- HEADER[relocBlockAddr, 8]
    P: blockSize <- relocBlock[4, 4]
    P: blockPage <- relocBlock[0, 4]

    P: firstEntryAddr <- ADD relocBlockAddr 8
    P: nEntry <- SHR (SUB blockSize 8) 1
    P: tmpSize <- SUB tmpSize blockSize
    P: nextBlockAddr <- ADD relocBlockAddr blockSize

    V99: NEq tmpSize 0

    P: tmpEntry <- INT 0 4
    V96: UGE nEntry 1
    L2(V96): entryAddr <- VLOOP(firstEntryAddr, nextBAddr, V98,
        10)
        P: entry <- HEADER[entryAddr, 2]
        P: tmpEntry <- ADD tmpEntry 1
        P: nextBAddr <- ADD entryAddr 2

        P: relocType <- SHR BITAND entry[1] 0xf0 4
        P: relocAddr <- BITAND entry 0xfff

    V11: EQ (BITAND (SHL one (SHR entry 12)) 0x3a0) 0 term
        V12: OR EQ relocType 10 ULE relocType 4 term
        V13: ULT ADD blockPage relocAddr imageEnd term

        ## RelocType 4 uses two entries instead of 1
        V14: Eq relocType 4
        P(V14): tmpEntry <- ADD tmpEntry 1
        P(V14): nextBAddr <- ADD nextBAddr 2

    ## From ntdll!LdrRelocateImageWithBias:47
    ### Checks that the RtlImageNtHeader is still valid
    ### Meaning that the targeted addre cannot be 0x0 or 0x1...
    V15: OR (EQ relocType 0) AND (NEQ relocAddr 0) (NEQ
```

```
        relocAddr 1) term
46      ### Cannot overlap with e_lfanew (0x3f-0x40)
47      V16: OR (EQ relocType 0) OR (ULT relocAddr 0x3f) (UGT
            relocAddr 0x40) term
48      ### Cannot overlap with PE magic (e_lfanew-e_lfanew+4)
49      V17: OR (EQ relocType 0) OR (ULT relocAddr HEADER.e_lfanew)
             (UGT relocAddr ADD HEADER.e_lfanew 4) term
50
51          V98: ULT tmpEntry nEntry
52      END L2
53  END L1
```

Listing A.2: Excerpt of the model of the loader of Windows 10 handling relocations

# Appendix B

# Summary of the Thesis in French

## B.1 Introduction

Au cours des dernières décennies, l'importance des systèmes informatiques dans les sociétés modernes a augmenté au point d'envahir tous les aspects de notre vie quotidienne. De nos jours, les systèmes informatiques constituent l'épine dorsale des infrastructures critiques, protègent nos informations personnelles et permettent la transmission rapide de données essentielles, ce qui affecte inévitablement notre vie privée, publique et politique.

Tout en apportant des avantages indiscutables, les technologies numériques ont également ouvert la porte à de nouvelles menaces qui ont souvent pris les utilisateurs et les vendeurs au dépourvu. Poussés par les gains potentiellement énormes que peut procurer l'exploitation d'un appareil aussi vital, des acteurs mal intentionnés ont commencé à cibler les systèmes informatiques ou à les utiliser pour perpétrer des fraudes.

L'une des stratégies couramment employées par les acteurs de la menace consiste à fournir un logiciel qui effectue des actions préjudiciables sur le système cible, souvent en le faisant passer pour inoffensif ou même souhaité. Internet a fourni un vecteur d'attaque sans précédent pour ce type de logiciel, au point que plusieurs auteurs ont comparé la propagation des logiciels malveillants à celle d'une épidémie.

Les parallèles entre le phénomène des logiciels malveillants et le domaine de l'épidémiologie ne s'arrêtent pas à l'ampleur de leur prolifération. Tout comme les bactéries et les virus, les logiciels malveillants ont tendance à s'adapter à leur environnement et finissent par contourner les barrières qui bloquent l'infection. En effet, malgré les efforts déployés par le secteur de la sécurité pour détecter, documenter et contrer rapidement les menaces émergentes, les auteurs de logiciels malveillants affûtent continuellement leurs outils, dans le but de dissimuler les comportements malveillants aux yeux inquisiteurs des analystes de logiciels malveillants.

La cause première de tous les types de mécanismes d'évasion est que les outils anti-malware doivent modéliser soit le système sur lequel le malware fonctionne, soit les capacités du malware lui-même. Par conséquent, les outils anti-malware ne raisonnent que sur une approximation du malware et de son exécution. Les divergences entre ces modèles et la réalité du comportement des logiciels malveillants ouvrent la voie au contournement des défenses et des analyses.

En toute justice, le secteur de la sécurité a acquis une expérience précieuse au cours de la course aux armements contre les logiciels malveillants qui a duré plusieurs décennies. Lorsque suffisamment de nouvelles

souches de logiciels malveillants échappant à l'analyse apparaissent, le secteur réagit en remettant en question les pratiques antérieures et en adoptant de nouvelles pour répondre aux nouvelles menaces.

Cependant, cette tendance aux approches réactives conduit à maintenir en place d'autres pratiques dont on sait qu'elles conduisent à des approximations lâches, ce dont les auteurs de logiciels malveillants peuvent tirer parti, du moins de manière théorique ou non évolutive.

Cette thèse remet en question les choix de conception des outils et techniques anti-malware, en adoptant et en promouvant une approche proactive de l'analyse des malwares. Nous nous concentrons en particulier sur deux pratiques qui sont encore très répandues, même si une bonne partie des praticiens les considèrent comme discutables. Contrairement à la croyance dominante, nous montrons que tirer profit de tels choix de conception n'est pas seulement un exercice théorique mais une menace concrète à laquelle l'industrie des anti-malwares n'est pas prête à faire face.

En termes plus spécifiques, cette thèse se concentre sur deux aspects des outils et de la recherche anti-malware modernes, omniprésents à la fois dans les pipelines d'analyse des malwares et dans les logiciels de protection des points finaux : l'utilisation d'informations au niveau des API pour coder le comportement des malwares, et la réimplémentation du chargeur de programmes du système d'exploitation.

Si les experts reconnaissent que les logiciels malveillants peuvent exploiter ces pratiques, ils affirment que c'est un mal nécessaire (et marginal). Cette thèse remet en question cet argument en montrant que tirer profit de ces pratiques est possible à grande échelle et de manière automatisée. En examinant les preuves récentes mises en lumière par les chercheurs en sécurité et la chasse aux logiciels malveillants dans la nature, nous démontrons également que les auteurs de logiciels malveillants montrent un intérêt croissant pour l'exploitation de ces pratiques. Enfin, nous étudions la possibilité de résoudre ces problèmes à la racine, en mesurant les difficultés que les architectes anti-malware peuvent rencontrer et en proposant des stratégies pour les résoudre.

Ce résumé (dont la structure reflète celle de la thèse elle-même) est organisé comme suit. La section 2 fournit les connaissances de base nécessaires à la compréhension du reste de la thèse. La section 3 est basée sur un article actuellement en cours de soumission à IEEE Security and Privacy 2022 et présente notre contournement générique du traçage des API ainsi qu'une caractérisation de l'écart sémantique entre les API et les syscalls, en se concentrant sur le système d'exploitation Windows. La section 4

présente notre travail "Exploring Syscall-Based Semantics Reconstruction of Android Applications", publié lors du 22e Symposium international sur la recherche en matière d'attaques, d'intrusions et de défenses, dans lequel nous tentons de reconstruire la sémantique de l'API à partir de traces de syscall enregistrées à partir d'applications Android [NBF19]. La section 5 est basée sur notre travail "Lost in the loader : The many faces of the windows PE file format" publié lors du 24e Symposium international sur la recherche en matière d'attaques, d'intrusions et de défenses, et systématise notre analyse des chargeurs PE dans l'écosystème Windows [NGFB21]. Enfin, la section 6 conclut le résumé et propose des orientations de recherche futures.

## B.2 Contexte

Compte tenu de l'ampleur du problème des logiciels malveillants, il n'est pas surprenant que les chercheurs et les praticiens aient développé de nombreux outils et approches pour le contrer.

Depuis le début du phénomène des logiciels malveillants, le secteur de la sécurité a mis en œuvre une stratégie en deux étapes pour atténuer son impact et protéger les utilisateurs finaux. À ses débuts, l'analyse des logiciels malveillants reposait essentiellement sur un travail manuel. Les analystes inspectaient des échantillons potentiellement malveillants, recueillaient des informations sur leur comportement et signalaient leurs caractéristiques distinctives s'ils étaient dangereux. Ces caractéristiques sont ensuite traduites en procédures exploitables - souvent appelées "signatures" - qui analysent les fichiers et les programmes de la machine de l'utilisateur et suppriment ceux qui correspondent à l'empreinte du logiciel malveillant.

Si ce modèle en deux étapes est resté en place jusqu'à présent, les technologies employées pour analyser les nouveaux logiciels malveillants et les détecter sur les équipements des utilisateurs ont évolué.

Du côté de l'analyse, la percée la plus importante a eu lieu avec l'introduction de pipelines automatisés qui génèrent des rapports sur plusieurs aspects de l'exécution des programmes, tels que les opérations du réseau et du système de fichiers et les modifications des paramètres du système. Les systèmes automatisés ont permis à l'analyse des logiciels malveillants de se développer et de suivre le rythme de l'augmentation constante du nombre d'échantillons découverts chaque jour. En fait, les analystes de logiciels malveillants peuvent désormais se concentrer uniquement sur les échantillons pour lesquels l'outil automatisé a signalé

quelque chose qui mérite d'être examiné, ce qui réduit les frais généraux liés à l'audit d'échantillons totalement inoffensifs.

L'étape de la protection des points de terminaison a également connu de nouveaux développements au fil des ans. Les outils de reconnaissance des signatures de logiciels malveillants ont commencé comme de simples scanners de modèles d'octets, mais ont fini par adopter des technologies plus puissantes, telles que les langages spécifiques à un domaine, pour effectuer des vérifications plus complexes. Par la suite, les sociétés antivirus ont introduit des signatures dynamiques (qui raisonnent sur les logiciels malveillants au moment de leur exécution) et des services de télémétrie pour collecter des statistiques sur la machine de l'utilisateur, à partir desquelles il est possible de graver des preuves d'attaques dissimulées. Codage du comportement du programme L'analyse dynamique (ou, comme on l'appelle souvent, comportementale) des logiciels malveillants vise à comprendre ce que fait le logiciel malveillant pendant son exécution.

Selon les aspects du comportement du logiciel malveillant sur lesquels l'analyse se concentre, les résultats de l'analyse contiennent différents types d'informations. Par exemple, la liste des adresses IP avec lesquelles le programme a établi la communication et leur contenu permettrait de dresser un tableau précis de son comportement du point de vue de la mise en réseau. D'autre part, la liste des fichiers créés, modifiés ou supprimés indiquerait le comportement du système de fichiers.

Une façon courante et plus générique de coder le comportement d'un programme consiste à suivre les API (Application Programming Interface) qu'il utilise. Les systèmes d'exploitation modernes fournissent ces interfaces de programmation hautement spécialisées qui facilitent le développement de fonctionnalités complexes utilisées par les logiciels bénins et malveillants.

Décrire le comportement d'un programme en termes d'API qu'il utilise est souhaitable pour de nombreuses raisons, la première étant la simplicité avec laquelle un analyste humain peut les interpréter. Cependant, il est bien connu que la collecte d'informations de niveau API de manière fiable n'est pas possible dans le cas général et que les logiciels (y compris les logiciels malveillants) peuvent éviter complètement d'utiliser les API de haut niveau. En s'appuyant sur des interfaces de niveau inférieur (ce que l'on appelle généralement l'interface syscall), les programmes peuvent réimplémenter complètement l'API de haut niveau, du moins en théorie.

Formats des fichiers exécutables Avant de lancer un programme, le système d'exploitation doit préparer un environnement adapté à

l'exécution du programme. Par exemple, des logiciels différents nécessitent des tailles différentes de mémoire privée et des configurations différentes de processeur pour fonctionner correctement. Les informations relatives à ces exigences sont codées dans les en-têtes du programme, c'est-à-dire des données structurées stockées au début du fichier exécutable, conformément à la spécification d'un format de fichier exécutable.

Les informations contenues dans les en-têtes du programme sont précieuses pour l'analyse des logiciels malveillants. Par exemple, les formats de fichiers exécutables codent la disposition en mémoire (appelée image) du programme ou, en d'autres termes, un ensemble d'instructions permettant de charger correctement le fichier du programme en mémoire. Savoir à quoi ressemble l'empreinte mémoire d'un programme au début de son exécution est un prérequis pour de nombreuses techniques antimalware avancées.

Cette thèse étudiera le format Portable Executable, le standard de facto des systèmes Windows. Étant donné le modèle de source fermée adopté par Microsoft, les implémentations du chargeur de Windows qui analysent et interprètent les données ne sont pas accessibles au public. Par conséquent, les responsables des outils de sécurité ont dû réimplémenter la logique d'analyse du format PE à partir de zéro, en se basant uniquement sur les spécifications du format.

Le plus souvent, cependant, ces implémentations personnalisées diffèrent légèrement les unes des autres et de celle fournie par le chargeur du système d'exploitation. Cela entraîne des divergences dans ce que différents outils (et même différentes versions d'un même système d'exploitation) considèrent comme un programme valide que les logiciels malveillants peuvent exploiter pour échapper à l'analyse et à la détection.

Si les travaux précédents ont mis en évidence plusieurs exemples de divergences entre différentes implémentations de la logique d'analyse syntaxique du PE, tous ont été découverts par essais et erreurs et non par une approche automatisée et systématique.

## B.3 Contournement générique et pratique du traçage des API pour les logiciels malveillants Windows

Ce chapitre présente notre exploration des problèmes qui surviennent lorsque les outils anti logiciels malveillants s'appuient sur des informa-

tions au niveau des API pour l'analyse comportementale.

En particulier, nous présentons une approche qui facilite remarquablement le processus de création d'exécutables élaborés capables de contourner l'analyse des logiciels malveillants basée sur les API. Cette recherche généralise certaines tendances récentes que les analystes ont signalées dans les nouvelles souches de logiciels malveillants qui invoquent directement les services de bas niveau du noyau du système d'exploitation (à savoir, les syscalls) dans le but de réduire leur empreinte API.

Notre système compile les programmes écrits en utilisant les API de haut niveau du système d'exploitation et fournit aux exécutables résultants le code implémentant les API dont il a besoin, dérivé des bibliothèques originales de Windows. En fait, au moment de la compilation, notre système résout les dépendances du programme et trouve les DLL dont il a besoin. Les DLL sont ensuite modifiées pour permettre au programme d'utiliser leurs fonctionnalités sans avoir besoin d'être chargées par le système d'exploitation, puis elles sont assemblées pour créer un environnement d'exécution personnalisé.

Au moment de l'exécution, les programmes utilisent les fonctionnalités fournies par le runtime embarqué au lieu des bibliothèques du système. Ce faisant, le programme est totalement transparent du point de vue de toute solution de surveillance des API. Ces outils, en fait, reposent sur l'hypothèse que pour qu'un programme puisse utiliser une API, son flux d'exécution doit atteindre le code fourni par les bibliothèques du système. Cependant, les programmes compilés avec notre approche n'ont jamais besoin d'entrer dans les modules fournis par le système (donc, potentiellement surveillés).

L'existence d'un tel mécanisme de contournement générique suggère que le fait de s'appuyer exclusivement sur des informations au niveau de l'API est préjudiciable à l'analyse des logiciels malveillants, car il ouvre la voie à l'évasion.

Au-delà d'une conception de l'analyse du comportement centrée sur les API, nous envisageons la couche syscall comme le meilleur poste de surveillance pour contrôler de manière fiable le comportement des programmes en cours d'exécution.

Cependant, l'utilisation des primitives syscalls comme blocs de base pour coder le comportement d'un programme a un prix. En fait, les appels système ont une sémantique beaucoup plus faible que les API, ce qui signifie qu'ils sont plus difficiles à interpréter pour les analystes humains.

De plus, certains aspects de l'exécution d'un programme peuvent ne

pas être accessibles du tout lorsqu'on choisit d'encoder son comportement en termes d'appels système. Certaines fonctionnalités que les programmes utilisent souvent, en fait, ne nécessitent pas l'intervention du système d'exploitation pour être réalisées. Les routines de cryptage et de décryptage qui ne nécessitent que des opérations mathématiques et qui, en tant que telles, peuvent être exécutées dans l'espace utilisateur sont des exemples notables de sécurité.

Dans la dernière partie du chapitre, nous rapportons les résultats d'une campagne d'analyse dynamique à grande échelle, qui caractérisent la complexité de l'écart sémantique entre les deux couches.

Pour chacun des plus de 23 000 programmes de notre ensemble de données, nous avons recueilli les traces API et syscall à l'aide d'un outil d'analyse dynamique personnalisé basé sur l'instrumentation binaire dynamique.

Nous avons ensuite analysé les données collectées pour comprendre quels aspects du comportement d'un programme peuvent être reconstruits à partir d'une trace de syscall, et la faisabilité de la récupération des informations au niveau de l'API.

Malheureusement, nos mesures sur les données collectées montrent que de nombreux facteurs rendent le problème de la mise en correspondance des syscalls et des API ambiguës, ce qui en fait un défi pratique.

## B.4 Exploration de la reconstruction sémantique des applications Android basée sur le syscalls

De manière similaire à la dernière partie du précédent, ce chapitre présente nos mesures sur la faisabilité de la reconstruction d'informations de niveau API à partir de traces de syscall, en ciblant cette fois le système d'exploitation Android.

À l'instar de ce qui se passe dans l'écosystème Windows, la grande majorité des cadres existants effectuent un traçage au niveau des API (c'est-à-dire qu'ils visent à obtenir la trace des API invoquées par une application donnée) et utilisent ces informations pour déterminer si l'application analysée contient des fonctionnalités indésirables ou malveillantes.

Suite à des travaux antérieurs qui ont montré que, dans Android, les mécanismes de traçage et d'instrumentation au niveau de l'API pouvaient être facilement contournés, quels que soient les détails de leur mise en œuvre spécifique, nous abordons dans ce chapitre le problème de la correspondance entre la couche API Java d'Android et la couche syscall Linux

sur laquelle elle repose.

La première partie de notre approche consiste à collecter des informations sur le temps d'exécution à partir d'un ensemble de données d'applications Android, de manière à pouvoir déduire les relations appelant-calculé entre les API et les appels système.

À cette fin, nous avons développé un nouveau système d'analyse dynamique capable de suivre à la fois les API Java et les appels système qu'une application Android invoque pendant son exécution.

Pour construire le système d'analyse, nous nous sommes appuyés sur des techniques d'analyse/édition automatique du code source pour doter le système d'exploitation Android d'une fonction de journalisation personnalisée. En particulier, nous avons instrumenté les points d'entrée et de sortie de chaque méthode Java dans le code source d'Android pour émettre un message personnalisé que l'interface de journalisation capture ensuite.

Pour faciliter le processus d'analyse de la quantité massive de données collectées pendant la phase d'analyse dynamique, nous avons créé une structure de données facile à interroger qui stocke des informations sur chaque API enregistrée, la base de connaissances.

Pour condenser les informations stockées dans la base de connaissances, nous avons créé des modèles pour chaque API enregistrée. Intuitivement, ces modèles fonctionnent comme des objets de type regex qui correspondent à toutes les traces d'appels système possibles que l'invocation d'une API peut produire. En tant que tels, ces modèles peuvent être utilisés pour trouver des preuves potentielles d'invocation d'API à partir d'une trace de syscall. Cependant, nos tentatives de reconstruire les informations sur les API à partir des traces d'appels système que nous avons enregistrées pendant l'analyse dynamique ont échoué en raison des similitudes inhérentes aux modèles des différentes API.

En essayant de caractériser les causes profondes, d'abord en examinant manuellement la base de connaissances, puis en utilisant une approche automatisée, nous avons découvert plusieurs modèles d'appels système qui apparaissent pendant l'invocation de nombreuses API différentes d'une manière apparemment non déterministe. Après une enquête plus approfondie, nous avons conclu que ces schémas proviennent des primitives de synchronisation et de gestion de la mémoire que de nombreuses API utilisent. Pour cette raison, nous considérons ces modèles de syscall comme du bruit puisqu'ils ne transmettent aucune information précieuse sur l'API qui les a invoqués.

Les modèles bruyants contribuent de manière significative à

l'ambiguïté de l'écart sémantique entre l'API et les couches syscall. Cependant, même en supprimant ces sources de bruit, la résolution du problème de la cartographie semble rester hors de portée. En fait, même avec des hypothèses fortes et favorables, les séquences d'appels système produites par les différentes API sont trop similaires pour être distinguées de manière fiable.

## B.5 Lost in the Loader : Les nombreux visages du format de fichier PE de Windows

Un problème connu dans le secteur de la sécurité est que les programmes qui traitent les formats de fichiers exécutables, tels que les chargeurs de systèmes d'exploitation, les outils de rétro-ingénierie et les logiciels antivirus, présentent souvent de légères divergences dans la façon dont ils interprètent un fichier d'entrée. Ces différences peuvent être exploitées par les attaquants pour échapper à la détection ou compliquer la rétro-ingénierie et sont souvent découvertes par les chercheurs par un processus manuel d'essais et d'erreurs.

Dans ce chapitre, nous présentons la première analyse et exploration systématique des analyseurs syntaxiques PE. À cette fin, nous avons créé un langage personnalisé spécifique au domaine afin de capturer facilement les détails sur la façon dont les différents logiciels analysent, vérifient et valident si un fichier est conforme à un ensemble de spécifications. Par conception, les modèles écrits dans notre langage peuvent être traduits en problèmes SMT dont les solutions sont des en-têtes PE que le parseur analysé accepte comme valides.

En tirant parti des propriétés mathématiques des problèmes SMT, nous avons développé un cadre qui exécute automatiquement diverses tâches qui seraient difficiles à réaliser manuellement. Par exemple, notre cadre peut produire des cas de test différentiels ; en d'autres termes, des en-têtes PE qu'une implémentation considère comme valides et que la deuxième implémentation marquerait comme non valides.

Nous avons ensuite utilisé ce langage personnalisé pour créer des modèles pour les chargeurs de trois versions de Windows (XP, 7 et 10) et les outils populaires radare2, ClamAV et Yara. La modélisation des chargeurs de Windows a été une tâche particulièrement difficile, car elle a nécessité une rétro-ingénierie approfondie de différents composants du système d'exploitation, du noyau aux bibliothèques dynamiques.

Au moyen de notre cadre, nous avons comparé ces modèles, en explo-

rant les divergences entre les implémentations de ces chargeurs. Par exemple, pour toute combinaison de deux versions de Windows, nous avons pu générer des exécutables PE qui s'exécutent sans problème sous la première, mais que la seconde rejette comme étant malformés. De même, nous avons généré des échantillons valides en fonction des fenêtres que les outils ont marquées comme non valides ou pour lesquelles ils ont fourni une cartographie mémoire inexacte.

Les résultats de notre analyse ont des conséquences sur plusieurs aspects de la sécurité des systèmes. Nous montrons que les outils d'analyse populaires peuvent être contournés, que les informations extraites par ces outils d'analyse peuvent être facilement manipulées et qu'il est trivial pour les auteurs de logiciels malveillants de prendre des empreintes digitales et de " cibler " uniquement des versions spécifiques d'un système d'exploitation d'une manière qui n'est pas évidente pour quelqu'un qui analyse l'exécutable.

Dans la dernière partie du chapitre, nous présentons les résultats d'une campagne de chasse aux logiciels malveillants que nous avons menée sur VirusTotal et qui visait à trouver des preuves de logiciels malveillants adoptant les divergences découvertes par notre cadre dans la nature. La campagne a trouvé plusieurs échantillons pour chaque divergence, ce qui suggère que les auteurs de logiciels malveillants s'efforcent de peaufiner leurs exécutables PE pour échapper aux analyses et aux défenses.

Plus important encore, ce travail montre que la fragmentation de la mise en œuvre des chargeurs de PE pose un réel problème. Comme nous l'avons souligné tout au long de ce chapitre, il n'existe pas une seule façon correcte d'analyser les fichiers PE, et même les différentes versions du système d'exploitation Windows traitent ce format de façon légèrement différente. Par conséquent, il ne suffit pas que les outils de sécurité corrigent les nombreuses incohérences que nous avons trouvées dans nos expériences, mais plutôt, pour s'attaquer au problème à la racine, ils devraient permettre à l'analyste de choisir lequel des différents modèles de chargeurs il doit émuler.

## B.6 Travaux futurs et conclusion

Les recherches présentées dans cette thèse suggèrent plusieurs pistes d'explorations futures dans le domaine de l'analyse des logiciels malveillants.

Bien que cette thèse fournisse des informations précieuses sur les caractéristiques du fossé sémantique entre les API et les syscalls, nous pensons que des efforts supplémentaires doivent être faits pour le combler de manière générique. En outre, si nous nous sommes concentrés sur les systèmes d'exploitation Windows et Android, d'autres plateformes doivent également être étudiées, notamment macOS et Linux. Ces deux systèmes d'exploitation posent des défis uniques. Par exemple, alors que Windows et Android prennent principalement en charge un ensemble d'API (l'API WinAPI et l'API Java d'Android), Linux ne dispose pas d'un seul langage de programmation (et donc d'un seul ensemble d'API). Cela signifie que pour récupérer des informations d'exécution de haut niveau d'un programme Linux à partir de ses appels système, il faut d'abord étudier le langage de programmation dans lequel le programme est écrit et ses API.

Le même raisonnement s'applique à la modélisation de différentes implémentations de chargeurs de programmes pour trouver des divergences. Il faut étudier davantage d'outils liés à la sécurité et davantage de versions de Windows pour comprendre de manière exhaustive l'ampleur du problème de différentiel d'analyse syntaxique pour le format PE. En outre, la prise en compte d'autres systèmes d'exploitation et formats exécutables représente également une voie viable pour de nouvelles recherches. Nous espérons que la méthodologie et les outils développés dans le cadre de cette thèse nous aideront dans cette tâche.

En plus des contributions techniques mises en évidence tout au long de la thèse, nous pensons que, sur un plan plus fondamental, le mérite de cette thèse est d'essayer d'anticiper les tendances futures des logiciels malveillants. L'étude et la remise en question des pratiques et des croyances courantes nous ont permis de découvrir de nouvelles voies d'évasion. En documentant notre travail de recherche et ses résultats, nous espérons sensibiliser la communauté de la sécurité aux lacunes de nos techniques actuelles et aux nouvelles menaces possibles. Pour paraphraser "L'art de la guerre" de Sun Tzu, seule la connaissance de soi et de son ennemi permet de ne pas craindre le résultat de cent batailles.

# References


[A. ] A. Albertini. Corkami PE files corpus. https://github.com/corkami/pocs/tree/master/PE.

[A. 13] A. Albertini. Making a Multi-Windows PE. *POC or GTFO*, (0x01), 2013.

[ABF+16] Vitor Afonso, Antonio Bianchi, Yanick Fratantonio, Adam Doupé, Mario Polino, Paulo de Geus, Christopher Kruegel, and Giovanni Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium*, pages 1–15, 2016.

[ADY13] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *Proceedings of the Security and Privacy in Communication Networks (SecureComm)*, 2013.

[Ale] Alexander Sotirov. TinyPE. http://www.phreedom.org/research/tinype/.

[Are] Areizen. Android malware sandbox. https://github.com/Areizen/Android-Malware-Sandbox.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[ASM+14] Daniel Arp, Michael Spreitzenbarth, Hubner Malte, Hugo Gascon, and Konrad Rieck. Drebin: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[B. 15] B. Baker, A. Chiu. Threat spotlight: Rombertik – gazing past the smoke, mirrors, and trapdoors. https://blogs.cisco.com/security/talos/rombertik, 2015.

[BB13] S Bratus and J Bangert. Elfs are dorky, elves are cool. *POC or GTFO*, (0x00), 2013.

[BBS+17] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber. ARTist: The Android Runtime Instrumentation and Security Toolkit. In *European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2017.

[BCL+07] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *USENIX Security Symposium*, page 15, 2007.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 265–275. IEEE, 2003.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.

[Bui] Hoang Bui. Bypass edr's memory protection, introduction to hooking. https://medium.com/@fsx30/bypass-edrs-memory-protection-introduction-to-hooking-2efb21acffd6. Accessed January 14, 2022.

[CGFB18] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding linux malware. In *IEEE Symposium on Security & Privacy*. IEEE Computer Society, May 2018.

[cho] Chocolatey, the package manager for windows. https://chocolatey.org/. Accessed January 14, 2022.

[CHY+16] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. Extract me if you can: Abusing pdf parsers in malware detectors. In *NDSS*, 2016.

[Cisa] Cisco. Clamav. https://www.clamav.net/.

[Cisb] Cisco. ClamAV - Bytecode Signatures. https://www.clamav.net/documents/bytecode-signatures.

[Cisc] Cisco. Clamav - file hash signatures. https://www.clamav.net/documents/file-hash-signatures.

[Cisd] Cisco Talos. Pyrebox, a python scriptable reverse engineering sandbox. https://blog.talosintelligence.com/2017/07/pyrebox.html.

[Cise] Cisco Talos. RATs and stealers rush through "Heaven's Gate" with new loader. https://blog.talosintelligence.com/2019/07/rats-and-stealers-rush-through-heavens.html.

[CMF+18] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 395–411, 2018.

[CML+21] Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. Obfuscation-resilient executable payload extraction from packed malware. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[Com] Comodo Cyber Security. Openedr. https://github.com/ComodoSecurity/openedr. Accessed January 14, 2022.

[Cyba] Cyberbit. Latest Trickbot Variant has New Tricks Up Its Sleeve.

[Cybb] Cyberbit. Malware Mitigation when Direct System Calls are Used.

[Cybc] Cyberbit. New 'early bird' code injection technique discovered. https://www.cyberbit.com/blog/endpoint-security/new-early-bird-code-injection-technique-discovered/. Accessed January 14, 2022.

[Cybd] Cyberbit. New LockPoS Malware Injection Technique.

[DAUR16] Marko Dimjašević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. Evaluation of android malware detection based on system calls. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics*, pages 1–8. ACM, 2016.

[Dev17] XDA Developers. Xposed installer (framework). http://repo.xposed.info/module/de.robv.android.xposed.installer, 2017.

[Dij75] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the International Conference on the Theory and Practice of Software, International Conference on Tools and Algorithms for the Construction and Analysis of Systems (ETAPS/TACAS)*, 2008.

[DQQY19] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. Decaf++: Elastic whole-system dynamic taint analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, pages 31–45, 2019.

[EGC+10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2010.

[EH06] Frank Ch Eigler and Red Hat. Problem solving with systemtap. In *Proceedings of the Ottawa Linux Symposium*, volume 2006, 2006.

[ELM16] Guillaume Endignoux, Olivier Levillain, and Jean-Yves Migeon. Caradoc: a pragmatic approach to pdf parsing and validation. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 126–139. IEEE, 2016.

[ero] erocarrera. pefile. https://github.com/erocarrera/pefile.

[ESE] ESET. LoudMiner uses virtualization software to mine cryptocurrency.

[FADA14] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-Based Detection of Android Malware Through Static Analysis. In *Proceedings of the ACM Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[fdr] F-droid — free and open source software applications for the android platform. https://f-droid.org/.

[Fir] Fireeye. Significant formbook distribution campaigns impacting the u.s. and south korea. https://www.fireeye.com/blog/threat-research/2017/10/formbook-malware-distribution-campaigns.html .

[FLBK15] Aristide Fattori, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. Hypervisor-based malware protection with accessminer. *Computers & Security*, 52:33–50, 2015.

[fri] Frida analysis framework. https://www.frida.re.

[GKP+15] Michael Gordon, Deokhwan Kim, Jeff Perkins, Limei Gilham, Nguyen Nguyen, and Martin Rinard. Information-Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *ACM Sigplan Notices*. ACM, 2005.

[Goo] Google. UI/Application Exerciser Monkey | Android Developers. http://developer.android.com/tools/help/monkey.html.

[GPJ17] Xinyang Ge, Mathias Payer, and Trent Jaeger. An evil copy: How the loader betrays you. In *NDSS*, 2017.

[GZZ+12] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[HB99] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Third USENIX Windows NT Symposium*, page 8. USENIX, July 1999.

[HBZ18] Nikolai Hampton, Zubair Baig, and Sherali Zeadally. Ransomware behavioural analysis on windows platforms. *Journal of information security and applications*, 40:44–51, 2018.

[HKFK18] Tobias Holl, Philipp Klocke, Fabian Franzen, and Julian Kirsch. Kernel-assisted debugging of linux applications. In *2nd Reversing and Offensive-oriented Trends Symposium 2018 (ROOTS)*, November 2018.

[HR] Hex-Rays. IDA Pro: a cross-platform multi-processor disassembler and debugger. http://www.hex-rays.com/products/ida/index.shtml.

[HTP15] Roee Hay, Omer Tripp, and Marco Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128. ACM, 2015.

[Hua06] Yinrong Huang. Vulnerabilities in portable executable (pe) file format for win32 architecture. Technical report, TR, Exurity Inc., Canada, 2006.

[Int] Intel. Pin a dynamic binary instrumentation tool. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html. Accessed January 14, 2022.

[J. 13] J. Bangert, R. Shapiro, S. Bratus. Weird Machines and revisiting Trusting Trust for binary toolchains. http://www.cs.dartmouth.edu/~sergey/trust/30c3-chain-of-trust.pdf, 2013.

[j00] j00ru. Windows system call tables. https://github.com/j00ru/windows-syscalls.

[java] Java documentation for javax.net.ssl.sslsocketfactory.createsocket. https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLSocketFactory.html#createSocket-java.net.Socket-java.lang.String-int-boolean-. Accessed: 2019-06-27.

[javb] Java Parser and Abstract Syntax Tree for Java. https://github.com/javaparser/javaparser.

[jpb] JPF-symbc: Symbolic PathFinder. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc.

[JS12] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *2012 IEEE Symposium on Security and Privacy*, pages 80–94. IEEE, 2012.

[KISH13] Yuhei Kawakoya, Makoto Iwamura, Eitaro Shioji, and Takeo Hariu. Api chaser: Anti-analysis resistant malware analyzer. In *International Workshop on Recent Advances in Intrusion Detection*, pages 123–143. Springer, 2013.

[KKB+06] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. Behavior-based spyware detection. In *Usenix Security Symposium*, page 694, 2006.

[KKD17] Doowon Kim, Bum Jun Kwon, and Tudor Dumitraş. Certified malware: Measuring breaches of trust in the windows code-signing pki. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1435–1448, 2017.

[KKK11] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 285–296, 2011.

[KKK15] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. A novel approach to detect malware based on api call sequence analysis. *International Journal of Distributed Sensor Networks*, 11(6):659101, 2015.

[KPS10] Dan Kaminsky, Meredith L Patterson, and Len Sassaman. Pki layer cake: New collision attacks against the global x. 509 infrastructure. In *International Conference on Financial Cryptography and Data Security*, pages 289–303. Springer, 2010.

[KSO+17] Yuhei Kawakoya, Eitaro Shioji, Yuto Otsuki, Makoto Iwamura, and Takeshi Yada. Stealth loader: Trace-free program loading for api obfuscation. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 217–237. Springer, 2017.

[KVK14] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: bare-metal analysis-based evasive malware detection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 287–301, 2014.

[LBK+10] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 399–412, 2010.

[LKC11] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 338–357. Springer, 2011.

[LMP+14] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[LNW+14] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor Van Der Veen, and Christian Platzer. Andrubis–1,000,000 apps later: A view on current android malware behaviors. In *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pages 3–17. IEEE, 2014.

[Mala] MalwareBytes. Floki bot and the stealthy dropper. https://blog.malwarebytes.com/threat-analysis/2016/11/floki-bot-and-the-stealthy-dropper/ .

[Malb] Malwarebytes. Process Doppelgänging meets Process Hollowing in Osiris dropper.

[Mica] Microsoft. Dynamic-link library search order. https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-library-search-order#search-order-for-desktop-applications.

[Micb] Microsoft. PE Format.

[Micc] Microsoft. Windows API sets.

[Mic18a] Microsoft. Control flow guard. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, 2018.

[Mic18b] Microsoft. LoadLibraryExA – Windows API. https://docs.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryexa, 2018.

[MMP+12] Nariman Mirzaei, Sam Malek, Corina S. Pasreanu, Naeem Esfahani, and Riyadh Mahmood. Testing Android Apps Through Symbolic Execution. In *ACM SIGSOFT Software Engineering Notes*, 2012.

[MOA+17] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[Mos] Fabian Mosch. A tale of edr bypass methods. https://s3cur3th1ssh1t.github.io/A-tale-of-EDR-bypass-methods/. Accessed January 14, 2022.

[MRG+18] Aravind Machiry, Nilo Redini, Eric Gustafson, Yanick Fratantonio, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Using Loops For Malware Classification Resilient to Feature-unaware Perturbations. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[MSF+08] Lorenzo Martignoni, Elizabeth Stinson, Matt Fredrikson, Somesh Jha, and John C Mitchell. A layered architecture

for detecting malicious behaviors. In *International Workshop on Recent Advances in Intrusion Detection*, pages 78–97. Springer, 2008.

[NBF19] Dario Nisi, Antonio Bianchi, and Yanick Fratantonio. Exploring syscall-based semantics reconstruction of android applications. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, Chaoyang District, Beijing, September 2019. USENIX Association.

[NGFB21] Dario Nisi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Lost in the loader: The many faces of the windows pe file format. In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2021)*, San Sebastian, Spain, October 2021. ACM.

[NW70] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[OM13] Digit Oktavianto and Iqbal Muhardianto. *Cuckoo malware analysis*. Packt Publishing Ltd, 2013.

[pe ] PE Format. https://docs.microsoft.com/en-gb/windows/win32/debug/pe-format.

[pev] pev - user manual. http://pev.sourceforge.net/doc/manual/en_us/.

[PHL+15] Radu S Pirscoveanu, Steven S Hansen, Thor MT Larsen, Matija Stevanovic, Jens Myrup Pedersen, and Alexandre Czech. Analysis of malware behavior: Type classification using machine learning. In *2015 International conference on cyber situational awareness, data analytics and assessment (CyberSA)*, pages 1–7. IEEE, 2015.

[rad] radare2, a portable reversing framework. http://www.radare.org/.

[RAMB16] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting Runtime Values in Android Applications that Feature Anti-Analysis Techniques. In *Proceedings of the*

*Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

[Red] Red Teaming Experiments. Apc queue code injection. https://www.ired.team/offensive-security/code-injection-process-injection/early-bird-apc-queue-code-injection. Accessed January 14, 2022.

[RFC13] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April*, 2013.

[roy] roy g biv / defjam. Virtual Code Windows 7 update. https://github.com/darkspik3/Valhalla-ezines/blob/master/Valhalla%20%233/articles/VCODE2.TXT.

[RSI12a] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows Internals, Part 1*. Microsoft Press, 2012.

[RSI12b] Mark E Russinovich, David A Solomon, and Alex Ionescu. *Windows Internals, Part 2*. Microsoft Press, 2012.

[RT20] Dima Rabadi and Sin G Teo. Advanced windows methods on malware detection and classification. In *Annual Computer Security Applications Conference*, pages 54–68, 2020.

[sau13] saurik. Exploit (& Fix) Android Master Key. http://www.saurik.com/id/17, 2013.

[SBS13] Rebecca Shapiro, Sergey Bratus, and Sean W. Smith. "weird machines" in ELF: A spotlight on the underappreciated metadata. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, Washington, D.C., August 2013. USENIX Association.

[Sec] Optiv Security. ScareCrow. https://github.com/optiv/ScareCrow.

[SFE+13] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 2013.

[Sig20] Siguza. Psychic Paper. https://siguza.github.io/psychicpaper/, 2020.

[ska06] skape. Locreate: An Anagram for Relocate. http://www.uninformed.org/?v=6&a=3&t=txt, 2006.

[SM07] Elizabeth Stinson and John C Mitchell. Characterizing bots' remote control behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 89–108. Springer, 2007.

[SWL16] Mingshen Sun, Tao Wei, and John C.S. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, CCS'16, 2016.

[T. 17] T. Petsios, A. Tang, S. J. Stolfo, A. D. Keromytis, S. Jana. NEZHA: Efficient Domain-independent Differential Testing. In *Proceedings of the 38th IEEE Symposium on Security & Privacy*, San Jose, CA, May 2017.

[TB15] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[TKFC15a] Kimberly Tam, Salahuddin Khan, Aristide Fattori, and Lorenzo Cavallaro. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[TKFC15b] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: automatic reconstruction of android malware behaviors. In *Ndss*, 2015.

[Tod17] Todd Cullum. Portable Executable File Corruption Preventing Malware From Running. https://toddcullumresearch.com/2017/07/16/portable-executable-file-corruption/, 2017.

[top] topjohnwu. Magisk. https://github.com/topjohnwu/Magisk.

[ule19] ulexec. ELF Crafting Advance Anti-Analysis techniques for the Linux Platform. https://github.com/radareorg/r2con2019/blob/master/talks/elf_crafting/ELF_Crafting_ulexec.pdf, 2019.

[UPGB19a] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. A close look at a daily dataset of malware samples. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):6:1–6:30, January 2019.

[UPGB19b] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. A close look at a daily dataset of malware samples. *ACM Transactions on Privacy and Security (TOPS)*, 22(1):1–30, 2019.

[vir] VirusTotal. https://www.virustotal.com/.

[Vir21] VirusTotal. File statistics during last 7 days. https://www.virustotal.com/en/statistics/, 2021.

[VP11] Mario Vuksan and Tomislav Pericin. Constant insecurity: Things you didn't know about portable executable file format. In *BlackHat*, 2011.

[WL16] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[XZGL14] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. Goldeneye: Efficiently and effectively unveiling malware's targeted environment. In *International Workshop on Recent Advances in Intrusion Detection*, pages 22–45. Springer, 2014.

[yara] VirtusTotal - yara in a nutshell. https://github.com/VirusTotal/yara.

[yarb] PE module — yara 4.0.2 documentation. https://yara.readthedocs.io/en/stable/modules/pe.html.

[YIT+16] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, et al. Sandprint: fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 165–187. Springer, 2016.

[YXA+15] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. AppContext: Differentiating Malicious

and Benign Mobile App Behaviors Using Context. In *Proceedings of the International International Conference on Software Engineering (ICSE)*, 2015.

[YY12] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, Bellevue, WA, August 2012. USENIX Association.

[ZWZJ12] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.